UNIVERSITY OF CALIFORNIA
SANTA CRUZ

FATE: A MORE EFFICIENT AND FLEXIBLE APPROACH TO DATA INTERPRETATION WITHIN PROTOCOL STACKS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

James Mathewson

March 2021

The Dissertation of James Mathewson
is approved:

J.J. Garcia-Luna-Aceves, Chair

Brad Smith

Katia Obraczka

Quentin Williams
Interim Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

FATE: A MORE EFFICIENT AND FLEXIBLE APPROACH TO DATA INTERPRETATION WITHIN PROTOCOL STACKS

by

James Mathewson

FATE (Functional Algebraic aTomic Evaluators) is introduced as an alternative to the traditional approach of communication protocols in which each protocol implements its own simple data interpretation module to decide the next steps to be taken by protocol agents executing the protocols. FATE uses algebraic expressions to evaluate information in a packet, message or stored piece of content and render a normalized scalar value that states the utility of the information contained in the data according to rules defined by the protocols using the data. This allows the aggregation of simple rules into very sophisticated expressions used to define the utility of the data. The approach advocated in FATE consists of aggregating the results of simple data evaluators (atomic evaluators) to obtain a utility value that reflects how useful the data are to the communication protocol or protocols using the data. This approach allows rapid development and simplifies the evaluation of results.

Because FATE uses equations to assign utilities to data, it allows a modular development and duplication of results by using the same formula to evaluate the same type of utility. This consistent behavior reduces protocol development time and effort when protocols have to be tested in simulators before being deployed in actual software and or hardware platforms.

Examples based on caching systems and forwarding of data packets are used to illustrate the advantages of using FATE as a plug-in data-evaluation engine of any protocol stack.

Acknowledgments

Much thanks to my advisor for his patience and support.

Chapter 1

Introduction

The approach currently used for the implementation of protocol stacks in computer networks dates back to the original development of the Internet more than 50 years ago. At the time the Internet was starting to evolve, computing and storage resources were scarce and expensive, which put major constraints on the amount of information that could be used by any given protocol agent, as well as the type of actions that any protocol agent could take to provide its services. As a result, communication protocols were organized into protocol stacks in which most protocols were decoupled from the physical medium, each protocol layer operated independently of other and communicated with the layer above and the layer below, and processing and storage inside the network were kept to a minimum. The resulting protocol designs and implementations attempted to minimize the use of processing and storage resources at routers, proxies, servers and clients while maximizing the utilization of network resources and ensuring that protocol actions were executed properly.

Given the processing and storage limitations that characterized the times when the basic Internet architecture was developed, it is not surprising that the algorithms used as part of protocol stacks were very simple insofar as interpreting

data is concerned. However, the availability of affordable processing and storage resources today has enabled the development of machine-learning algorithms capable of interpreting the same pieces of data in different n ways to derive useful information from the data.

This thesis introduces *Functional Algebraic aTomic Evaluators* (**FATE**) as a new approach for the implementation of protocols and protocol stacks that enables far more flexible and effective use of data by means of data interpreters (called "experts" in machine-learning approaches) that evaluate and rank pieces of data according to predefined policies defined as part of a given protocol or mechanism used in a protocol stack.

Multiple data interpreters can be used on the same piece of data with each interpreter giving a ranking based on a normalized scalar from 0 to 1 to the data, and a protocol-dependent policy module aggregates the rankings from the interpreters to determine the action that a protocol agent should take based on the result given by the policy module. Results provided by data interpreters need not be confined within a protocol layer; they can be shared across the protocol stack and among multiple hosts and routers by proper tagging of messages and packets shared within a system or among systems.

FATE can be viewed as a *plug-in data-interpreter engine* that replaces the simple data-interpreter routines used in traditional protocol implementations. Each such engine consists of one or multiple data interpreters and one or multiple aggregators of rankings, with a final aggregator or rankings providing a signal to protocol agents regarding the rewards or steps that the protocol agent should take.

The novelty of FATE is not the use of data interpreters as an integral part of a communication-protocol implementation, because any such implementation must

have a data interpreter that informs a protocol agent of any rewards derived from prior steps take by the agent, or steps required by the agent based on the data that were evaluated. The novelty of FATE derives from introducing a systematic approach for the use of a wide variety of data interpreters that can act on the same or different pieces of data to inform protocol agents of rewards and needed next steps based on protocol policies. By requiring that each data interpreter use a function that ranks the fate of some piece of data with a normalized scalar, FATE allows multiple interpreters based on very different functions to provide valuable inputs regarding the fate of data. Furthermore, the normalized rankings used in FATE enable the use of policies with which rankings can be aggregated to take into account a wide variety of performance requirements and administrative constraints.

FATE eliminates the complexity of designing data interpreters for different protocols based on multiple criteria and constraints by using multiple evaluators for well defined subsets of the criteria and constraints that must be taken into account, and implementing each evaluator with a simple algebraic formula (configurable via XML) that can be verified to reflect the intent of the design. As a plug-in data-interpreter engine, FATE can supplement existing protocols or replace them, depending on the designers' intent. As an example, FATE can be used as the data-interpreter engine for packet forwarding of IPv4 to extend traditional IPv4 forwarding with evaluators that allow a router to make forwarding decisions based on the information carried for multiple protocol layers in each data packet as well as the perceived conditions of different routes to the intended destination. Parameters such as quality of service, hop count, remaining battery life of wireless routers, link and path bandwidth, congestion, and others can be taken into account through multiple atomic evaluators and a policy-based aggregation of their

rankings.

FATE can provide more intelligence to the operation of a single protocol, multiple protocols, or an entire protocol stack. Although the use of reinforcement learning and other approaches to machine learning (ML) are outside the scope of this thesis, the design of FATE can enable a more systematic use of ML techniques in protocol stacks without the need to change the protocols themselves, which will be apparent to the reader after our description of a few examples of how FATE works in concrete communication protocols.

The use of algebraic equations to express the functions of data evaluators (e.g., (0.8*LRU+0.2*QoS)*FRESHNESS) means that the same outputs are attained given the same inputs over any implementation platform. Consequently, FATE can be implemented in exactly the same way in ns-3 or NSNSim simulators, or a linux box. The only difference in results is due to the simulator or environment (e.g., ns-3 may not simulate noise with sufficient accuracy) rather than the implementation of data evaluators. FATE allows algebraic formulas to communicate the exact method being used over heterogeneous platforms.

This dissertation focuses on the application of FATE to two key areas in computer networks, namely: distributed caching and routing and forwarding. Caching is currently a research area receiving considerable attention because of its importance in making content delivery over the Internet more efficient. There has been considerable research on distributed caching systems over many years; however, all prior and existing approaches to distributed caching assume very specific content replacement policies and algorithms (e.g., least frequently used) simply because they are well known or simple to implement., rather than the efficiency with which they can adapt to user requirements or network constraints. Applying FATE in caching systems can help address user interests and system requirements in an ef-

fective manner. Each individual FATE data evaluator (expert) provides a ranking reflecting the utility of the content, and multiple evaluators may be weighted and combined to provide the desired evaluation.

Applying FATE to forwarding functionality is also a timely example of its importance in the implementation of modern protocol stacks. Routing with multiple constraints has been addressed by many authors in the past. However, little attention has been given on the use of similar statements of constraints or policies applied to different types of networks, like ad-hoc networks, disruption-tolerant networks, server-area networks, or the Internet. FATE allows simple, quick, and safe changes to a routing system, without having to redesign a routing infrastructure from scratch. FATE can consider hops, power, bandwidth, buffer bloat, and the like for the purposes of determining the cost of a path, and it can consider constraints by changing an algebraic formula used as part of a route-computation module. In addition, FATE is stateful, and uses packets to communicate information, not only of the data, but the paths used for data delivery. This flexibility allows congested nodes to communicate their state, and allow other forwarding nodes to make adjustments as necessary.

Chapter 2 summarizes prior work on areas related to new network architectures related to different aspects of FATE. The main difference between this prior work and FATE is that FATE is designed to provide flexibility within the context of an existing protocol stack, rather than defining a new one, and it uses the same approach for the interpretation of data and extraction of their utility at all layers of the stack.

Chapter 3 presents the basic architecture that defines FATE. FATE is written in C++11, and adapted to the ns-3 simulator. Packet formats are based on named-data (data type) format that allows easy creation of packet fields. Each

node is represented by multiple modules, each designed to be modular and fulfills a specific action (e.g., caching or forwarding). FATE uses a shim layer to interface from its environment to the current environment; as an example a timer can use the simulation time when used with ns-3, or Linux timer when run on the linux platform.

Chapter 4 describes a simple implementation of FATE aimed at improving cache hit rate. The implementation uses two different cache-eviction algorithms, LRU and LFU, and gives a weighted combination to give a more efficient cache rate than either of them individually, but retains the temporal properties of LRU, which plagues LFU (e.g, highest request content changes over time). This simple example of two content ranking algorithms gives superior performance over either individual component (LRU or LFU) and illustrates the advantages enabled by the flexible data evaluation engine introduced in FATE.

Chapter 5 covers a method to increase LFU cache hit rate at the expense of tracking a few more entries. Storage can be a costly resource in some edge systems storing very large files consuming most of the storage space. Using ethereal LFU, the requests are monitored beyond what is in storage, e.g., track 250 items, but only have storage for 50 items. This example shows that a much higher hit rate can be achieved by extending the state to track evicted data than by only monitoring the content in storage. Ethereal LFU uses a significantly smaller amount of memory to give a noticeably higher cache hit rate.

Chapter 6 gives greater insight on the flexibility of FATE by applying it to non-traditional QoS caching. Each individual QoS property is cached and then combined. Each evaluation is easily combined with other using FATE even though they are normally not compatible with one another. This is attained thanks to the use of algebraic formulas that provide data utilities as outputs. The constraints

chosen for this FATE example are file size, QoS (e.g., ranking), regex matching (highly important messages), distance to producer, and LRU. These are considered individually and then combined one at a time to show the power and ability to take into account multiple, traditionally incompatible, constraints using FATE.

Chapter 7 handles the problem of cache locations in an information-centric network (ICN) by using a distributed cache network. A content store may be placed in the network, but not be available on-path. This results in lowered content cache efficiency. Even if the 'cache everywhere' principle is applied, most of the on-path cache nodes have the same content, which is a waste of resources. To resolve this, FATE combines forwarding and caching, to create a hashed-cache topology. Even a single hash redirect to a cache node greatly improves network performance over a missed on-path network. The network uses fewer hops by routing content to a hashed cache (and producer if missed), over a non-cached path, which is shorter. The increased hit rate reduces the total number of hops taken, resulting in better content latency delivery to customers. The efficiency of a large single cache is compared to multiple, smaller caches.

Chapter 8 shows that FATE can evaluate egress packets using various criteria, By using these criteria, packets pending on a buffer are evaluated and transmitted according to priority and best egress QoS qualifiers (e.g., requirements for hop count, network bandwidth, etc).

Chapter 2

Prior Work

FATE is based upon ICN framework, keeping the information (data and metadata) with the packet. Information Centric Networking (ICN) is an architectural solution to the perceived deficiencies in the current Internet architecture. In Information Centric Networking (ICN), access to data is based by name (named-data), not by an end point or machine addresses. ICN treats content as king, and reception of Content is what is important, not where the content is located. There are several variants of ICN, but the common underline design philosophy is Information should be accessed by name, not by a machine address.

Unfortunately, each of the ICN based architectures have their own naming convention, various methods to implement security, caching-routing algorithms and architectural dependencies. In addition, improvements to ICN are hampered by the hard coded designs and official implementations, especially in various simulators (e.g. NDN ref 2.3.2), which make a solution more difficult to derive. Others, such as CCNX (ref 2.3.1) define packet fields, but lack discovery protocols. The use of terms like NDO (Named Data Object), Content, Information, and Data packet are equivalent and used interchangeably in this document.

Compounding the simulation process is the difficulty in implementing code

specific to each architecture and simulator. Simulator code for ns-3 (ref 8.2.1) does not work with ccnSim [25], which uses omnet++ [101]. Algorithmically, NDN and CCNX, which have a common base, have now diverged in the manner of interest response, again, complicating comparisons. These problems do not even consider different networking constraints, the actual algorithmic goal (for caching/routing), nor the modifications necessary to modify a protocol to cover corner cases.

Prior work includes ENCODERS (sec 8.2.2), which was the first implementation of FATE (called utility networking, at that time). ENCODERS implemented utility caching, utility prefetch, and a limited (social cluster orientated) version of utility forwarding. Most of the interesting work in this area came from using social hierarchies for packet routing, Delay-Tolerant Networks, and caching within pocket networks to avoid content extinction[52, 106]. The problem with encoders was its implementation was tied to an Android architecture, and was based upon HAGGLE[87], which limited its flexibility. Other related work includes ICN (2.3, DTN (2.1, various caching placement architectures (some specific to their architecture) (2.4.1, and various caching algorithms (2.4 for evicting low quality content.

2.1 ICEMAN-ENCODERS

FATE implements significant improvements and changes in architecture from encoders (ref 8.2.2, adding Utility Security, improved utility forwarding, and greatly improved interoperability within and between various nodes (lessons learned from the prior work). But, at this time, Encoders still has many features, such as Utility Forwarding (with node popularity identification and destination prediction), Utility Caching support for Network Encoding, and a Utility Caching Content Prefetch Predictor, as presented in ICEMAN[107, 105, 52]. Encoders,

using ICEMAN (via Utility Networking) originally supported the various DTN cooperative caching models[23], but was extended to model social networks by using the node popularity to effectively disseminate data[106]. Encoders was run on a virtual platform, using CORE[14] and EMANE[51], and mobility models supplied from BonnMotion[17].

2.2 Protocol Engines

FATE shares many similarities with CAPE (ref [44]). FATE and CAPE are both context aware protocol engines, both are capable of sharing context information via packets, share a similar TLV packet format (like NDN/CCNX), spans all protocol stacks (non hierarchical) and FATE allows signaling/data packets to be integrated (similar to CAPE). FATE extends upon context aware networking, as it also includes caching, and its actions are user configurable. CAPE nodes share context rich packets to other nodes, and make a forwarding decision based upon them. FATE also shares context rich packets to other nodes, but the packet metadata may be used in caching and/or forwarding decisions. CAPE is a new layer 2 protocol, while FATE can reuse existing stack protocols (and their metadata). FATE is more flexible, it can be a new packet format, or be encapsulated by MAC/IP/UDP/TCP packets. This allows rapid development, for work on the network layer (or function, in the case of caching) that is desired. FATE uses an algebraic formula of weighted algorithms, to make decisions. For caching, these can be relegated to cache the content, evict content, or ignore the content. For forwarding, FATE allows evaluation of packets to be transmitted, to properly weight the priority. As an example, buffer bloat can be avoided, by sending the most valuable packet (via predefined QoS parameters), and dropping expired packets in the buffer (e.g. after 'x' seconds). FATE forwarding is flexible, in that the

packets are evaluated by packet evaluation (QoS), cost of destination via each 1-hop neighbor (e.g. hops), or consideration of transmitting for new constraint (e.g. routing packets via node battery life, or based upon a valid time constraint for satellite transmission). FATE can easily handle new constraints by adjustments of its algebraic formula; distance, packet size, QoS value of packet, age of packet (has it expired?), bandwidth, method of wireless connectivity (bluetooth, wifi, or wired). FATE can be configured for a wide variety of conditions. FATE can evaluate and use the context rich content in the packets. Another difference, FATE is configured via XML, and allows a large variety of decisions making based upon algebra from weighted algorithms (e.g. LRU, size, battery power, or location); while CAPE excels in layer 2 MANET routing (reducing network contention, and increasing throughput). In short, CAPE is a scalpel for MANET routing, while FATE is a generic toolbox to resolve a variety of problems in networking. FATE is similar to greedy algorithms, which may not give the most optimal solution, but it allows rapid testing and evaluation to give a very good solution.

2.3 ICN Architectures

There are several variants of the ICN architectures[20, 100, 109], but the focus will be on CCNX and NDN. Initially, CCNX (version 0.73) and NDN (version 0.1) were the same except for a few minor naming differences. Continuous development after creative differences have resulted in the current versions, CCNX 1.0 (active at PARC) and NDN 0.3.2 (led by UCLA).

2.3.1 Content Centric Networking (CCN)

CCNX[75], with the latest version of 1.0, substantially changes the message format from the prior version. Instead of two messages (Interest and Data), this model has three message types (Data, Interest, and InterestResponse). The consumer will request a named-data-objects (NDO), referred to as Content in CCNX, via an Interest message. The message will travel to the network, until a match to cached/stored data is found. Typically, any node may have the requested Content, and send back a Content message, containing the requested content. If the content is not cached, it will be forwarded to the Producer of the content, and, if the Producer has the content, it will deliver the Data message. If the Producer does not have the content, or is unable to deliver content (e.g. Server busy or off-line), then an InterestResponse message is transmitted.

CCNX models each message type (Content, Interest, InterestResponse) as a hierarchical TLV packet, with a fixed Header. CCNX uses a PIT (Pending Interest Table) to record interests and the associated ingress port. When a Content packet or InterestResponse packet is received, it matches (via name) which interface requested the Content, and transmits it, via the PIT, to the correct interface. CCNX also uses a FIB (Forwarding Information Base), to egress Interest packets to the appropriate destination. A Content Store (CS) is an alias for a caching module. CCNX relies on securing Content, not end points. As such, unsolicited content is not stored in the CS, to avoid cache poisoning[45], which removes support for opportunistic caching.

Some of the deficiencies of CCNX lie in the lack of optimizing caching algorithms and a lack of discovery protocols (used in the FIB). In addition, there is no actual CCNX specific simulator (but there is CCNX implementation code), with the exception of NDNSim v1 (using CCNX 0.73)[11] and SCoNet[73].

SCoNet stands for Simulator COntent NETwork[73] was a heavily modified NdnSim (sec 2.3.2), which supported CCNX 1.0. Amongst its features was full TLV typed support and Interest-Content-InterestResponse packet support.

2.3.2 Named Data Networking (NDN)

NDN[112, 98, 11, 67] is an Send-Receive based architecture, where a request (Interest) packet is sent, and a corresponding response (Data) packet is sent back. NDN calls 'information' as Named-Data-Object (NDO), as each chunk of data literally has a unique name. Unlike CCNX, which has an InterestResponse packet for undeliverable data, NDN sends back a Data packet with a return code. The current code base is at version 0.3.2, which corresponds to NdnSim v2.1. Similar to CCNX, it supports a PIT, FIB, and Content Store.
NDN development is in parallel with NdnSim, discussed below.

NdnSim NDNSim (version 1) [11] was released in June 2012. It covered both NDN version 0.1 and ccnx version 0.7.3, being written in c++, python, and using ns-3 simulator. It had partial NDN feature support (packet attributes), but a full implementation of the NDN architecture, with fully functional Content Store, Pending Interest Table, and Forwarding Information Base. It had a very simplistic native caching system which resolved to a dual inheritance scheme of either (freshness, probability) with an (lru, lfu, fifo, random) caching scheme. Caching is based upon a trie structure, resulting in returning the *closest* match, not exact match. Unfortunately, it did not have a full packet implementation, with many features not enabled.
Version 2 [67] was released on January 27, 2015, and is a substantial upgrade. NDNSim v2 uses the NDN-cxx library, which allows the same code to be used in the simulator, as well real applications. This gives it a more realistic simulation.

It also supports caching from version 1, as its own caching code was being developed. The problem with both versions, is the tight coupling and implementation, making algorithmic changes difficult.

Version 2.1 was released on September 4, 2015, with the most notable change being caching, and supporting NFD's RIB manager. It is fully NDN 0.3.2 compliant. The caching architecture was updated to a skip list, and natively evicts first unsolicited packets, then stale packets, and finally packets in a fifo basis.

The most recent versions of NDN and NdnSim have the latest architectural changes, which differs significantly from CCNX v1.0, these are:

1. Caching-Selectors : Hits to an interest packet are not to an exact match, as in CCNX. NDN will return the *closest* NDO match. The interest packet defines 'selectors', which include several fields to further refine how to match data.
2. SuffixComponents are a range of valid numeric appendage matching (e.g. name match with a '0' to infinite, will match the request) at the end of the name.
3. Exclude to exclude all specified matches. Multiple Excludes are allowed, and many times, required, for the correct NDO.
4. ChildSelector to limit child matches. If '/NDN' is requested, all NDO's with the prefix of '/NDN' are matched (e.g. '/NDN/pics/' or '/NDN/logs/'). By using selectors, you limit the response.
5. TLV format - NDN uses a variable length TLV scheme, which can be from 1 to 8 bytes. Each TLV is typed to a specific TLV, unlike CCNX which uses a relative hierarchical TLV type.
6. Repository - Caching support similar to Content Delivery Networks (CDN). It differs from the CS by having larger storage and being more persistent.

7. Sync - This architectural component, SYNC, derived from the need to have peer-to-peer NDN applications. Specifically, Chronochat is a peer to peer Ndn client for video conferencing.
8. Data Muling - Support for Data Muling allows Vehicular MANET support, to cache (opportunistically) data from other nodes.
9. Forwarding Strategy module - Another architectural addition, to decide when, and where, to forward packets. This is the Named Forwarding Daemon (NFD) used in NdnSim.
10. NLSR - NDN Linked State Routing. NDN routing is done via NLSR, which populates FIB for forwarding.

2.3.3 NS3

NS-3[31] is an event driven simulator, using C++ and python code to schedule events. It is one of the largest supported simulators, with over thousand papers published on its platform, and the platform NdnSim uses for its base simulation. The strength of NS3 is it is event driven, and allows multiple processes to be run on the correct platform, thus giving results in a significantly faster time frame.

2.3.4 Other ICN Architecture

There are various examples of other ICN architectures[59, 38] , such as: Psirp-Pursuit[42, 7, 6], Comet[1], Haggle[87], PAL[5] and Sail[8]; all of which are European funded forays into ICN. All of these were funded under the FP7 European Union grant for Future Internet Assembly (FIA), and seem to be defunct with their main sites not updating since 2013. The funding for XIA[2] was under NSF grants CNS-1040757, CNS-1040800, and CNS-104080, with published research into 2014.

ICARUS [83] is a python based caching simulator for ICN based architectures (focused on CCNX and NDN), using named-content, request-response model (e.g. Interest, Content requests), supporting various protocols for evaluation. Icarus supports both cache placement strategies (where to place copies of content) and eviction strategies.

CAKA [80] is a PUB-SUB ICN architecture which exploits path diversity and cache awareness to deliver content.

ccnSim [25] is a ccn packet level simulator based upon OMNET++[101].

NETINF [13, 34, 61, 35] (NETwork of INFormation) is an ICN model supporting both peer caching, on-path and off-path caching. It supports both consumer-subscriber and DHT (PUB-SUB) distributed models.

GreenICN [3, 61] focuses on disaster scenarios using low power devices. and large scale video, and summarization (e.g. you do not need 10000 of the same reports).

CASCADE [61, 58] was a competitor to ENCODERS in the DARPA CBMEN (Content-Based Mobile Edge Networking) program. It uses a standard PUT-GET methodology, built over TCP and UDP, assuming groups of nodes (community) are stable.

2.3.5 ENCODERS

ENCODERS[61, 95] (Edge Networking with Content-Oriented Declarative Enhanced Routing and Storage) is an SRI implementation of the PSIRP ICN models,

based upon Haggle[87]. PSIRP and Encoders both use a bloom filter based Pub-Sub model, disseminating interests to neighbor nodes, and those neighbors return matches to the data. Even more generic than NDN's selectors, the interest request represents matches to desired attributes of a file (e.g. pic=cats, format=jpg), with each attribute weighted for matching. The bloom filter was also disseminated throughout the network, to match requested content, and identify which nodes may have the content. Matches were then shared (pushing content to the requesting node), unless the receiving node explicitly denies the packet.

Encoders was created to support the Darpa CBMEN program. Encoders target platform are android devices, and it is capable of disseminating and caching packets and entire files. Encoders used a heavily modified Haggle core[87], but more importantly, it used a prior iteration of Utility Networking (Utility Caching-Prefetch, and Utility Forwarding). Due to an early cancellation of the program, work was transferred to the simulation environment (as opposed to a real hardware system) for better scale of testing. Forwarding was configurable, and was done via Prophet[66], Direct[89], or Epidemic[99].

2.4 Caching Algorithms

The primary focus of this research is on caching, but caching is now intertwined with routing (prefetching or distributed cooperative caching)[82] and security. For the purpose of focusing on ICN, it is about the named content, which puts focus on where to cache, and what content to evict. Currently, most popular Internet traffic can be modeled by a zipf distribution, with cache effectiveness increasing with a corresponding logarithmic increase in cache size[19].

2.4.1 Cache Placement

[12] Cache placement assumes more than one node is capable of caching (which ICN does), and optimizes where the content is placed for optimal access.

1. ProbCache[78] - Probabilistic In-Network Caching recognizes the problems with on-path caching, which creates unnecessary redundancy. ProbCache records cache capacity along the data path, giving each node weight. It also records a TimesIn factor (derived from the distance of the caching node), and multiplies both together, giving the cache a probability of caching the content. From a Utility stand point, this will be broken down into two separate utilities (Cache Capacity and HOP Distance From Consumer).

2. LCD[62, 63] (Leave Copy Down). LCD caches Content at the $(l-1)$ level cache, or one cacheable level closer to the requesting client, from where the hit occurs. It requires multiple hits to bring the Content to the leaf cache.

3. NRR[26, 53] (Nearest Replica Routing) is the technique to access cached content, possibly off path. Typically, a PUB-SUB type methodology would use NRR.

4. Opportunistic[53, 27] - Caching from opportunity, when content is available. Certain methods, such as WAVE[27] have the server send extra chunks along with the requested content, marking the content as 'suggest to cache'. As opposed to prefetching, where a cache asks for extra chunks, the server provides the extra chunks, so it may be opportunistically cached. Another form of opportunistic caching algorithm is DITTO[36], which uses on-path caching, and content 'overheard' by the node (though the content was not directly routed through the node itself).

5. CDN[85, 65, 22] (Content Delivery Network) are dedicated commercial servers to host high demand content. Typically, a CDN will host static content (such as ads), and the host company may provide dynamic content to be served by the CDN. According to Chai[22] and Li[65], the proper placement of caching nodes (like CDN placement) offers the most benefit for the least cost.

6. Cooperative[71, 79, 22] - Caches cooperate by calling each other for requested Content, as opposed to immediately passing the request to the server. Based upon a paper by Wolman[104], cooperative caching is more effective on low edge populations than larger edge populations.

7. DHT[28, 96, 88] (Distributed Hash Table) use a hash key to locate or access the content. Typically used in peer-to-peer systems. An example of this is bit torrent[30] and CHOORD[96, 108].

8. Edge Caching[32, 46, 82, 41] The most efficient and effective caching is at the edge, and on-path opportunistic caching offers little improvement. Primarily, the edge cache will provide sufficient caching and more cache hits, due to the zipf characteristic of Internet traffic, while out-lying caching nodes will have significantly less hits.

9. Popularity[49, 106, 18] Popularity, not in content (which traditionally follows a zipf distribution), but popularity of nodes in a DTN. It is believed more popular nodes (by virtue of meeting more nodes), is the ideal location for content caching. Usually, popularity deals with social communities, as a few nodes tend to be members in multiple communities, making them prime candidates for content ferrying and caching.

2.4.2 Cache Eviction

When a cache becomes full, it is necessary to evict content, to make room for new content. A partial list of eviction strategies:

1. FIFO - First In, First Out. Content is sorted per cache entry time.

2. RAND - Content is randomly chosen and discarded.

3. LRU - Least Recently Used Content is evicted first.

4. LFU - Least Frequently Used Content is evicted first. Typically has an associated counter.

5. LRFU[64] - Least Recently Frequently Used. A mathematical implementation which can act as either LRU, LFU or both, based upon parameters λ. It strength comes from needing only a single past access time and is based upon the weighted function $F(x) = \left(\frac{1}{2}\right)^{x\lambda}$ equation, where λ represents a weight towards spatial (LFU at 0) and temporal (LRU at 1), or a combination if weighted between (0,1). The actual LRFU calculation is based upon $C_{tk}(b) = C_{t_{k-1}}(b) \times F(t_k - t_{k-1}) + F(0)$, where t_k is the current time and t_{k-1} is the last reference time.

6. FRESHNESS - Some content may be labeled with a 'freshness' value[75], whereby, after a period of time, the content is no longer valid, and becomes 'stale'.

7. COST[16] - Hit-miss ratios may not be the determining factor of a cache, but the cost of a CDN to host the content. In summary, it is more cost effective to host many popular files, than fewer higher-popularity files (especially with respect to size of hosted files).

2.5 Mobile Ad Hoc Networks, Delay Tolerant Networking

An Ad Hoc network is a collection of nodes, which communicate over a decentralized network. There is no pre-existing configuration or infrastructure for communication. Ad Hoc networks are noted to be self configuring, with devices joining/leaving the network at any time. Ad Hoc networks may be further constrained by mobility (MANETs) and range. VANETs (Vehicular Ad Hoc Network) use communication between various vehicles and possibly, roadside equipment. They are categorized by having high mobility (ranging from vehicles to jets), and may be used for traditional data communications, military, and intelligent self driving vehicles. SPAN (Smart Phone Ad Hoc Network) are typically used in cellular phones, where the end point is mobile, but the infrastructure is static. All MANET variants are very susceptible to links being established, broken, and re-established to varying nodes in the network. Delay Tolerant Networking (DTN) is characterized by intermittent network connectivity, possible asymmetric connectivity (one node can hear the other node, but not vice versa), poor reliability, and potentially large delays. Opposed to MANET, which form their own network, DTN have additional constraints on connectivity to deliver data, such as a need to connect to a sporadically available external network (eg communicating via a drone, satellite, with a small time frame). DTN has a more unique set of problems, such as which content to transmit in a short period of time, which content should be stored in the meantime, etc. Some DTN architectures use bloom filters to summarize their cached content, as done by haggle-encoders (ref 8.2.2) and DiPIT[110]. Other DTN architectures exist[57, 92, 94, 93, 86]

Chapter 3

FATE

3.1 FATE Architecture

The key objective in the design of FATE is to make the necessary step of interpreting data more efficient and flexible than is possible today in traditional protocol stacks. In the context of a protocol stack or a communication protocol, data can be obtained from the environment, local storage or processes executing in the same host, or other protocol agents running in remote systems.

To serve the above purpose, FATE is organized as a plug-in data-interpreter engine that can be easily integrated into the normal operation of existing protocols or in the design of new protocols. Each instance of FATE is composed of inter-linked modules, with each module having a specific purpose intended to either evaluate data according to a particular formula or aggregate multiple evaluations according to a predefined policy. Communication among FATE nodes is handled by a TLV packet format to be defined in a subsequent subsection and which can be standalone or appended to an existing protocol, such as UDP/TCP packets. Each module is specific to a task and evaluates the utility of the information contained in the data carried in a packet, message or stored piece of content. Based

on this utility the module may pass the evaluation to another module or act on the information.

Each module that evaluates the utility of data is called a data evaluator or *expert*. Such an expert has as its input a piece of data and its output consists of a normalized scalar value between 0 and 1 that states the utility of the data it received as input. The mapping of the input data to the utility value is defined by a specific function defined for a given a protocol to work correctly. To make the deployment of FATE instances simpler, each expert is defined using simple functions that need not capture the entire utility of a piece of data but are simple to implement, verify and reuse. However, the outputs of multiple experts can be combined with one another to implement more complex functions in much the same way as simple propositions can be combined into more complex propositions describing complex statements.

As shown in the example of Figure 3.1, multiple experts (data interpreters) may exist for the same purpose. In the Figure, determining if a content object may be reused has two experts (e.g., LRU or LFU). FATE allows multiple expert systems for the same evaluation or for different purposes to be combined by aggregators into a single evaluation.

In the case of caching, the main actions taken over a piece of content is to evict or to keep it. However, the utility of the cached content is dependent on what is the desired result. The following outcomes are a small example of desired caching outcomes:

1. Maximize Availability of Important Content: Important content may belong to a commercial customer, who pays for the privilege of higher availability.

2. Minimize delay for the consumer: Most 'valuable' content closer to edge

3. Minimize content fetched from producer

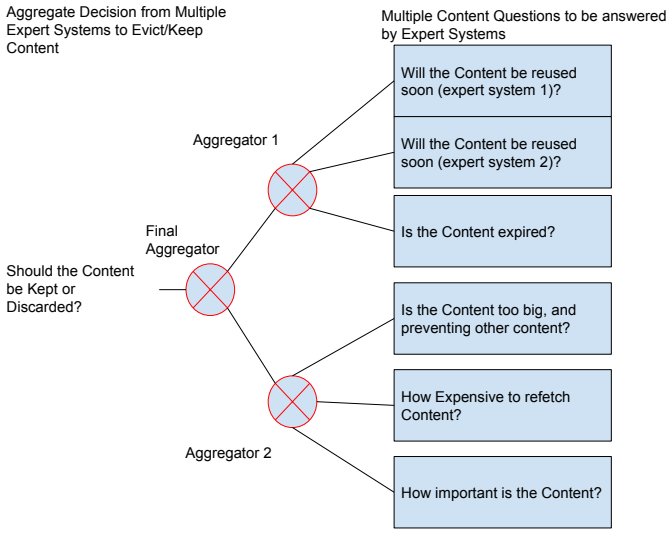

Figure 3.1: Overview of aggregation of multiple answered questions from various expert systems

4. Maximize content availability

5. Maximize high-priority content for distribution in a network.

Traditional Internet caching uses a single algorithm, typically least-frequently used (LRU), to decide which content piece stays or goes. But this methodology is only good when all content, all customers, are considered equal, which is hardly the case in reality. For the military, it may be important to have availability of content, in case their infrastructure is damaged. For organizations like CERT (Community Emergency Response Team), limited communications may require long term caching of critical requests (e.g. medicine, medical attention, notification of family, over movie streaming). Using FATE in the context of caching, some experts for caching can operate to maximize hit rate; others can consider QoS settings, the size of packets, freshness of content, the cost of retrieving content from the producer, and the like.

In the example shown in Figure 3.2, three egress ports in a packet switch or

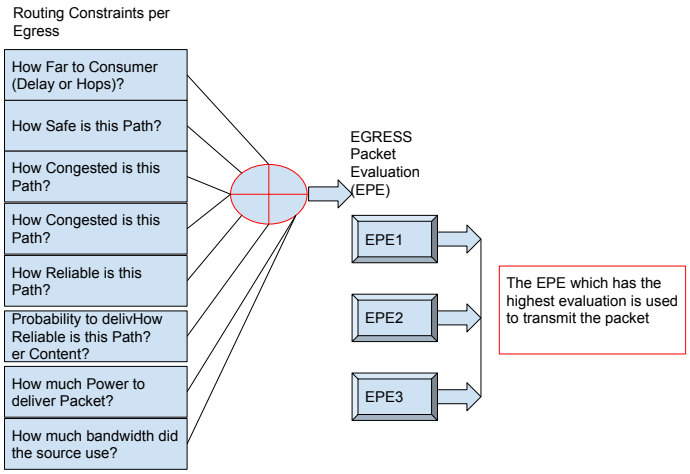

Figure 3.2: Overview of aggregation of multiple answered questions from various experts in forwarding

router evaluate a packet against various criteria, and the port with the highest evaluation is selected. The same principals can be used to minimize buffer bloat, congested paths, or minimize (shaping-sharing) a network port from a consumer IP to minimize excessive bandwidth use. Since there is no layered stack, no information is lost; FATE can use a source IP address to determine bandwidth usage, MAC address, port numbers, TOS fields, and FATE specific fields.

FATE modules can share state among them within the same host or across links or a network, and uses dedicated network discovery packets to share information among nodes. In general, discovery packets are network wide, and using non discovery packets to share information relates to specific events well defined within existing protocols.

3.2 FATE TLV Packet Format

FATE uses a type-length-value (TLV) packet format because of its flexibility and malleability of packet content. We chose to adopt a superset of the CCNX model [91] with non-predefined types and extended packet purposes and unique name support. FATE TLV packets do not have a fixed header, like CCNX, but the same information is encoded in a pure TLV packet format, adding the fixed header fields as additional TLVs contained in the packet. Another implementation difference is that CCNX uses relative type values to dictate which TLV it is, e.g., a CCNX TLV type of '1' may be Name TLV or Data TLV, depending on what the identity of the parent is. FATE does not have a hierarchical structure, and uses a flat format that is used to identify the field using absolute types. FATE supports both a binary encoded TLV packet format, or an ASCII XML format used in the packet. The packet format can be stand alone, or encapsulated in another format (e.g., TCP or UDP) packets. FATE operates on TLV's by matching either an enumerated value or a name. Each module may evaluate a packet, and mark the results of the evaluation by creating a temporary attribute on the packet. This is useful to allow other modules to use the same packet, and pass evaluations. As an example, for a cache-hit (or miss), a field is added "CACHEHIT" with a value of '1.0' for cache hit, and '0.0' for a cache miss. Other modules may use the temporary attribute fields for evaluation (e.g., IPv4 TTL or TOS field) or perform an action.

FATE can use its own TTL field, but it is just as easy to use the TTL field built into IPv4. All Layer 2/3/4 fields are translated into temporary FATE field attributes. This is to allow reuse for pre-existing packet attributes (e.g., IPv4 TOS field), but you are free to create new attributes. In addition, NS3 uses a socket method to send/receive packets. All the Layer 3 (and layer 4) information is

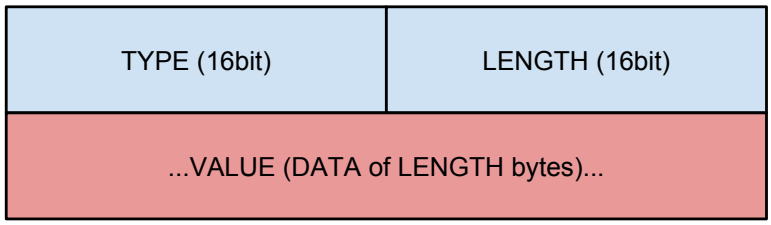

Figure 3.3: Basic TLV format

lost, which makes cache implementation impossible. To do otherwise will require extensive modifications to NS3, and is not a viable solution. Allows multiple intermediate destinations (consumer->cache-> producer), to allow cache hashing. FATE is agnostic over any protocol. To ensure independence over any protocol constraints, the protocol format is saved as temporary attributes (MAC, IPv4/6, TCP/UDP, or token ring). This allows FATE to work independent of any protocol.

3.2.1 TLV Packet Design

TLV packets are packets with a field for type, a field for length (of data), then a value (or data/content) field of the specified range. NDN defines type and length to be 1-8 byte variable length fields, using their values to encode the exact length[67]. CCNX [90] and FATE define both fields to be a fixed 16 bit format (Fig 3.3). The format is determined by the type. Unlike CCNX and NDN, FATE does not use any header format. In general, there are three categories of data types: Packet Name (which may contain heirarchy of other TLV's), Raw Data (string or serial byte format), and Endian Data (the value depends on the architecture). Each are discussed below, with implementation details.

FATE [73] is based upon the CCNX TLV format, and breaks it down into three generic types, with the following methodology and implementations.

String/Name TLV Name TLV is a one of the three main TLV primitives. Typically, string terminated TLV are used in the full name of the packet, which is parsed (internally) to allow content name matching.

Endian Data TLV Endian data TLV supports endian formatted data in sizes of 8 bytes. All data is written in network format, and read back in native host format. Eight bytes is chosen as the default, as it can easily values of 1,2,4 bytes.

Raw Data TLV RawData TLV is the last of the three main TLV primitives. Its value field is a byte vector. It supports raw byte pointer, string object, and vector STL formats. In addition to these types, it contains a template function to SetData and GetData, for any object specified. These template methods support both shallow copy methods, and the c++ redirect (string formatted) methods, allowing any object, using these methods, to be copied into the Value field. As a simple example, a double data type can be set into the TLV. A shallow copy will copy the entire variable (as long as all nodes use the same architectural endian format, this is acceptable), but the c++ string formatted redirect uses an object's overloaded << and >> to preserve its value.

Sample Packet Implementation Fate handles the details of converting names to more efficient values. In NS3, FATE automatically registers the name, to produce a unique numeric value. In real world purposes, the values are saved and converted, and must be applied to each node. Standard C++ calls, as shown in listing 3.1, are all that is necessary to populate a packet from a programming direction.

```
PktType Pkt; //Create packet
//Add any packet named-attribute (name, value, field-type)
//e.g. (stringname, any-value, bool temporary-field)
Pkt.SetUnsignedNamedAttribute ("uint64_t", 5555ULL, false);
```

```
Pkt.SetNamedAttribute ("double", 55.55555, false);
Pkt.SetNamedAttribute ("string", "jim:55.55555", false);
Pkt.SetSignedNamedAttribute ("int64_t", -5556LL, true);
//What kind of packet is it?
Pkt.SetPacketPurpose (PktType::DATAPKT);
testSt_t testStruct;
testStruct.a = 5;
testStruct.b = -3.14;
testStruct.c = 'c';
//memcopy is a shallow copy example
Pkt.SetObjectCpyNamedAttribute ("memcopy", testStruct, true);
//printcopy is uses C++ overloaded "<<" and ">>" operators
Pkt.SetPrintedNamedAttribute ("printcopy", 12.3456, false);
//remove fields or all fields
Pkt.DeleteNamedAttribute("memcopy", true); //remove memcopy temp attrib
Pkt.DeleteAllAttributes(false); //remove all permanent attributes
//ICN is named-packet, so add a unique name
IcnName < std::string > nextName = "/test1/test2/PacketName1";
//add an extra qualify, now its /test1/test2/PacketName1/part=2
nextName.SetUniqAttribute ("part", 2);
Pkt.SetName (nextName); //Put packet in name
```

Listing 3.1: Packet Implementation

3.3 FATE Implementation

FATE introduces several concepts to assist rapid development and testing of network protocols and algorithms. First, the code is written in C++11 that is agnostic of the platform (ns-3, Qualnet, linux, and others). To resolve system dependencies The code is written to use various resources, such as a timers, which are wrapped around the model; in other words, on a linux platform, the linux timer is called; on an ns-3 simulator platform, its native timer is called. Second, a flexible packet framework, which uses a type-name-value tuple in the packet, discussed in section 3.3.3 is used. FATE does use uniquely named information, or named data, to identify each unique chunk of data.

3.3.1 Functional Algebraic aTomic Evaluators

The concept of FATE is to evaluate information (typically via named packets of information), and perform an action, based upon the result. In order 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], which allows each function, no matter how dissimilar in evaluation, will always return a normalized value, which can be compared or evaluated with each other. Since all functions return a normalized scalar, actions are based upon matching an expected range. Some content may be cached, but only high-value content should be cached. In this example, and arbitrary value of (0.3, 1] (exclusive of 0.3, up to, and including 1.0), is eligible for caching. In other cases, the highest value is used. In this document, one or more functional algebraic atomic evaluators are used for a purpose (e.g. caching decision), or called *'Utilities'* (such as utility caching).

Aggregation Functions

FATE supports several aggregation methods, all take one or more inputs, and returns an appropriate result. Below is a partial listing of available aggregation functions:

1. MIN : MINIMUM(a,b,..,z) returns the minimum value of its inputs.
2. MAX : MAXIMUM(a,b,..,z) returns the maximum value of its inputs.
3. SUM : ADDITION(a,b,..,z) returns the sum of all its input. The sum may

be greater than 1.0, and may require scaling.

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

5. IF..THEN..ELSE : IF (function, range) THEN (function2) ELSE (function3). The *'function'* is evaluated, and if within the specified *'range'*, the value of *'function2'* is returned. If 'function' is not within the specified range, return the value 'function3'.

6. StepFn : StepFunction(function,range) if *'function'* has a value with the specified range, return a value of 1.0, otherwise return a value of 0.0.

Atomic Functions

FATE supports several atomic methods. For some functions, they have an option to rank information based upon configuration settings (such as LRU, which may be evaluated temporally or spatially). Each atomic function may be stateful, but the state is exclusive to each instance of the function. Atomic functions 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 principal to have many singular functions do the evaluation, then weighted based upon the appropriate aggregate function).

The following is a subset of atomic algebraic functions, currently available in our FATE implementation:

1. 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 uses Normalizers (3.3.2) to dictate how different LRU times will be parsed in relation to each other. As shown in Figure 3.4, which

shows values from 0, 1, 3, 8, and 10 seconds earlier, the different evaluations between Normal and Step Ranked values is shown.

2. 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 by different normalizer methods. As shown in Figure 3.5), and like LRU, normal or step 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.

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

4. HASH : HASH(rawdata, modulus) uses a modulus type function, based upon a hash value of either the name, name attribute, data, or an attribute. If the modulus of the hash matches a configured value, it returns a 1.0, else it returns a 0.0.

5. COUNT has several options to increment, decrement, or no change. Typically, increment is used to measure hop counts, decrement is used for a traditional packet TTL field, and 'no change' mimics a TOS (type of service) field.

6. ATTRIBUTE returns a match based upon the name field. At configuration, the match can be an exact match, a partial match (for string based data) or a range match (for numeric data).

7. FRESH returns an evaluation for fresh, or non-stale cache content. Since the data is temporal (how many milliseconds it is fresh), when it is transmitted, the FRESHNESS field is decremented by how long it has been in the node. Content which is fresh for ten seconds, and gets a cache hit four seconds later, will send out a FRESHNESS field valid for six seconds (10-4=6). The value returned is dependent on the normalizer, but typically will be of the impulse or linear degradation (see Figure 3.6) form.

8. NAME does an exact name match, typically used in producer-consumer end nodes. It returns a '1' on a match, otherwise '0'.

9. NameAttrMatch looks at a field in the name of the packet, and checks if the integer value is within a specified range. Helpful to match segments to a specific range.

10. U64Eval takes an integer value and returns a normalized value, based upon all integer values recorded.

11. PLE protect last element. This returns a value of '1' for the last element inserted. Useful for functions, such as LFU, to keep the last inserted item safe from purging.

12. NormEval is a normalized scalar value, and evaluated at the same valuation. Typically, another evaluator in another module evaluates the named-content, and writes it as a temporary attribute in the packet. This allows other modules to use the same evaluation. Examples include a normalized security evaluation, used in caching and forwarding modules.

13. RegexMatch uses a regex match of a specific packet field on the value.

14. NAMECHAIN does not evaluate, per se, but appends current node name to the data. Useful in path tracing of a packet, or discovering caching nodes along a path. When used in an algebraic formula, it returns a predefined (at configuration) constant value (with the exception of how it updates a packet, acts effectively as a constant).

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 may use different Normalizers, based upon how the algorithm is defined, to use approximate a temporal or weighted implementation metric.

Some of the choice on which implementation of an algorithm depends on possible memory or computation intensity for said algorithm. Normal ranked, which is temporally based, can quickly calculate the values; whereas Step ranked content (spatially biased), requires a calculation of order complexity $O(n)$ to find the desired value, biased from the lowers/highest values.

Functions such as FRESHNESS are always evaluated, as it is a comparison of when the content was first received, and how long the content will remain fresh. In contrast, RANDOM also has a secondary evaluation method: Upon receipt and upon evaluation. Upon receipt, the NDO is given, via a random number generator, a value which will not change. Upon evaluation, each time the content is evaluated, a new random number is provided. The first implementation requires state, while the second method does not require any state.

Utility Block Function

A single evaluator, Utility Block, has a unique function. Every chain of functions, in each module (8.3.2), cumulates, or passes its value to the Utility Block

function. The Utility Block function receives the evaluation from the Utility functions, and tags the packet with a temporary attribute (3.4.4), which is the result of the evaluation of said module. Using this methodology, a security evaluation may be tagged to a packet, which can be used in caching or forwarding module evaluations.

3.3.2 Normalizers

Many utilities will take a whole number (such as integers, timestamps, or hop counts), and evaluate all in relation to each other to produce a normalized scalar. This allows any utility to standardize on normalization. Many times, the value is desired to be inverted (or $(1 - val)$), depending on what is being evaluated (e.g. distance vs size), and is shown on abbreviated evaluations as "i{value}" .

1. *NormalMatch* will match a range of values, and if true will return '1' otherwise it will return 0. Abbreviated with "_Nm[lowMatch,highMatch]"

2. *GeometricRanked* values as 1/n. There is a 'biaseLowVal' option, to bias the return value from the lowest rank (e.g. 5,19,24 become 1/(5-5+1), 1/(19-5+1), 1/(24-5+1). Abbreviated formula is "_Ngb{bias}"

3. *NormalRanked* values are taken as a difference. The options are 'ceiling' (val-min)/(max-min) (option 'c'), 'fullRange' (val/max) (option 'r'), and 'floor' (val-min)/(max-min) (option 'f'). The abbreviated formula is "_Nro{option}". Normal Ranked content is typical of temporal parsing. The information is evaluated when it was received, in relation to other information. As depicted in the graph 3.4, information received 0,1,3,8, and 10 seconds prior, the information is evaluated at: 1.0, 0.9, 0.7, 0.2 and 0.

4. *StepRanked* return equal stepped values (e.g. 1,2,4 return as 1, 2/4, 1/4). The abbreviated formula is "_Ns".

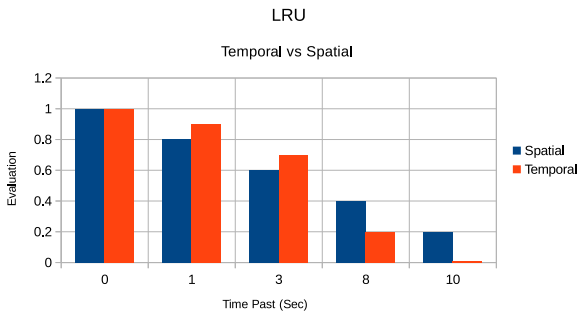

Figure 3.4: Normal vs step ranked values

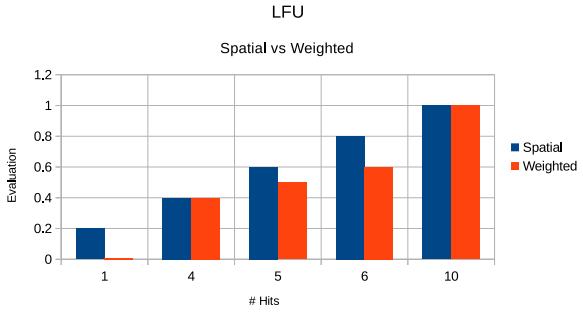

Figure 3.5: Inverted normal vs step ranked values

3.3.3 FATE Packets

Fate packet format resembles CCNX, in its implementation, using the TLV (Type-Length-Value) format. FATE is not a layered protocol. It can be encapsulated into IP packets, or be a standalone protocol.

Packet Fields

Fate is very flexible in the types of information embedded into the packet. Each attribute name can be a custom name based upon a TLV (type-length-value)

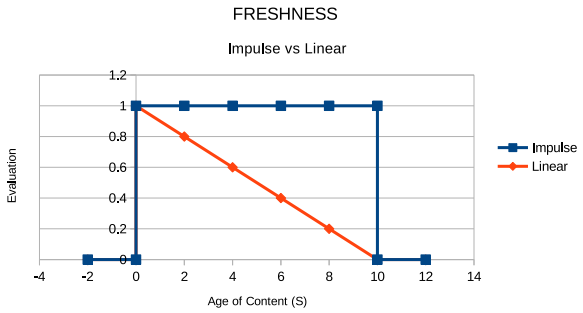

Figure 3.6: Freshness impulse vs linear values

packet format, unlike CCNX or NDN, which has most of the packet attributes predefined. In addition, FATE allows various means of embedding data:

1. String: Utf-8 character array.
2. Unsigned Integer: Unsigned Integer from 8 to 64 bits
3. Signed Integer: Signed Integer from 8 to 64 bits
4. Raw/Binary Data: Array of raw/binary data.
5. Floating Point: Double/floating point value. Not network endian safe.
6. Shallow Copy: Shallow copy of any object/structure. Pointer values are not copied.
7. Stream-String Conversion: Convert an object/structure to its equivalent string stream format (e.g. cout << myObject), transmit it as a string, then convert back into a valid object (e.g. cin >> myObject).

The data can be acted upon by a function, evaluated by a utility, or meant solely for an end user.

3.4 XML Configuration

An example node configuration is presented in listing 3.2. In the example, an XML file configures three modules (caching, security, and forwarding). The caching module is associated with the 'CacheStore', and stores up to 16 packets. The caching module uses a 'LRU*LFU' utility to evict low value content.
The security module, arbitrarily, evaluates the security of the information. As an example, the utility security evaluates a traditional TTL (using COUNT) implementation, and the utility block *tags*, with a temporary attribute, the results to the packet under evaluation. If a packet does not have a 'TtlHop' field, this module is configured to add the field, with a default value of '10'. This allows FATE networks to ensure a packet meets its expectations for evaluations.
Finally, the forwarding module, which uses IPv4 tags to route, adds its name to the packet, to track its progress. Finally, a store configuration (used with cache). It should be noted the caching module has its own 'size', and the store has its own 'size'; they are not required to be equal. This allows us to make decisions on more Information, even if the store is unable to hold all the content desired.

3.4.1 Packet Types

Each module only acts on specified packet types. A Caching (or Content Store) module will use both Interest and Data packets for evaluation; with Data packets updating the Utilities (new content), and an Interest packet may refresh or change the value of content (e.g. LRU or LFU). Since FATE packets are flat, a Discovery Module may use all packet types to refresh the status of its nearest neighbors, such as an interest or control packet refreshing a timeout in a next-neighbor hop table, while a Discovery or DiscoveryResponse packet may add new entries in said

```
<NodeModule>
  <UtilityModule moduleName="CacheBasicManager"
      associatedStore="CacheStore" cacheSize="16" >
    <Utility name="UTILITYBLOCK" >
        <Utility name="MULT">
            <Utility name="LRU" />
            <Utility name="LFU" />
        </Utility>
    </Utility>
  </UtilityModule>

  <UtilityModule moduleName="SecurityBasicManager" >
    <Utility name="UTILITYBLOCK" proxyName="SecurityBlock" >
        <Utility name="COUNT" missing_count_value="10"
         matching_lower_bound="0"
                  matching_upper_bound="1" count_condition="decrement"
                  match_criteria="LeftRightInclusive" attribName="TtlHop"
                      />
    </Utility>
  </UtilityModule>

  <UtilityModule moduleName="ForwardNs3Ipv4Manager" >
    <Utility name="UTILITYBLOCK" >
        <Utility name="NAMECHAIN" appendNodeName="true"
         m_defaultAttribute="Path:" nodeNamePartition=" ; "
                  appendIfnExist="true" appendInFront="false"
         nodeNameUnique="false"/>
    </Utility>
  </UtilityModule>

  <Store storeName="CacheBasicStore" name="CacheStore" size="16"
    storageMethod="MemMap"/>
</NodeModule>
```

Listing 3.2: Sample Node Configuration

table. A potential list of packet types are:

1. Interest : Packets request information, based upon their name field.
2. Data : Packets contain named data.
3. DataResponse : If an Interest packet can not be satisfied, the DataResponse packet returns any relevant error codes or additional information to the correct location (e.g. Data is moved or been renamed).
4. Control : Packets are used to control the Node or it's modules. Configuration can be done through Control packets
5. ControlResponse : Response to control packets, if it is expected.
6. Discovery : Packets used for network discovery, defined by a users algorithm (e.g. HELLO packets)
7. DiscoveryResponse : Response to Discovery packets, if expected.
8. Debug : Packets used to ferry debug commands, either through the network, or an explicit command for the node to log specific events or data.
9. DebugResponse : Packet response from the network version of Debug packets. Contains response to desired Debug commands.

3.4.2 XML And Binary Compact Serialization

FATE packets can be represented in three simple formats: Native, XML, and Compact Binary Serialization. The native format is used within the C++ environment, with easy access to each packet metadata attribute. XML representation

is a simple XML format, used in both ascii display of a packet, and can be used to represent a packet, add fields, or remove fields, based upon a white-black-red list. The XML representation is typical of control-configuration packets. Each node can be configured by a configuration file, XML or native Control packet, to change basic functionality. Binary Compact Serialization is used to send the packet on a wire, with a known network endianness.

3.4.3 Packet Name

FATE, based upon an ICN architecture does require each NDO to have a unique name to identify the corresponding unique content. Fate, like CCNX, allows a flat or named heirarchy, to help identify the location of the NDO. In addition, qualifiers are allowed, with support to act upon name, qualifier or both. No specific qualifiers are reserved, but matches are setup at the consumer/producer level. An example name, with qualifiers can be /fate/sample.jpg/seg=5/vers=3/res=320x200 or sample.jpg_seg5_vers3_res=320x200. By allowing qualifiers, different functions can act on the path, the name, or any of the qualifiers (e.g. Hash).

3.4.4 Metadata

Two types of meta-data are supported in a fate packet: Temporary and Native. Native meta-data is an intra-attribute of the packet itself. As the packet is passed, Native meta-data follows the packet.

Temporary meta-data is only valid inside a node, and is used to record results from one Utility, to be evaluated by another Utility. This is done to minimize expensive computation, especially of the same function throughout a node. Temporary can also be information and attributes for other protocols. As an example, FATE can be delivered by IPv4, but transitional meta-data for layer 2/3/4 is added for

```
<FATEPKT purpose="20" name="/test1/fileNum=2/segment=0">
  <Attribute name="CacheHit" nameType="16" dataType="4" data="1" />
  <Attribute name="CacheHitNodeName" nameType="19" dataType="1"
    data="Node2" />
  <Attribute name="NAMECHAIN" nameType="15" dataType="1"
    data="Node2 ; " />
  <Attribute name="ServerHitNodeName" nameType="18" dataType="4"
    data="0" />
  <Attribute name="Timestamp" nameType="2" dataType="2"
    data="94357700000000" />
  <Attribute name="TtlHop" nameType="1" dataType="4" data="127" />
  <TempAttribute name="SecurityEval" nameType="4" dataType="4"
    data="0.667" />
</FATEPKT>
```

Listing 3.3: Sample of Packet XML format

evaluation. When the packet is transmitted from a node, transitional meta-data is removed (like temporary data) and used to create the correct carrier packet. An example of all the meta-attributes is shown in listing 3.3.

3.4.5 Packet Examples

To help facilitate understanding of a FATE packet, the XML representation of it shown below. It shows all three attribute types being used (e.g. A security module gives the packet a value of .667, and attached a temporary attribute to the packet, for further evaluation).

3.4.6 White-Black-Red Attribute List Packet Transformation

Typically, when a Data packet is stored in the Information Store, or when a server replies to an Interest packet, the attributes of a packet may change. As an example a hop count (TTL) field is meaningless to a cache store (it should add a new hop count, instead of using a reduced count from the server). Likewise, a hop count may be useful to measure distance between consumers and producers.

When a a data packet is stored in the information store, or when a server returns an interest packet as a data packet, both are cases of the attribute fields being removed, changed, or added (beyond what the consumer expects).

White lists are the attribute names to keep as packet attributes; all other attributes are removed. Black lists are the attribute names to remove as packet attributes; all other attributes are retained. Red lists are new attributes to add (or overwrite) in a packet. By using this methodology, the user can define his protocol, which attributes to keep, and which to remove. Typically, this only applies to data objects upon caching, and interest (transformed into data) objects at the producer.

3.4.7 Metadata Name Registration

During serialization, only the name type and data is used. The name type associates both the name and data type being used. In order to keep the types and numeric equivalent representations in synchronization for each node, the string names are registered to a unique integer value. In ns-3, this is accomplished in a global function, where a named attribute or field is registered, with the correct integer representation returned. For a non global system (such as actual network nodes), this is accomplished with a predefined synchronized attribute file. Typically, the global registration is easiest and used in simulation. Once the named attributes are known, they are saved in a configuration file and configured into each node, to ensure correct attribute interpretation.

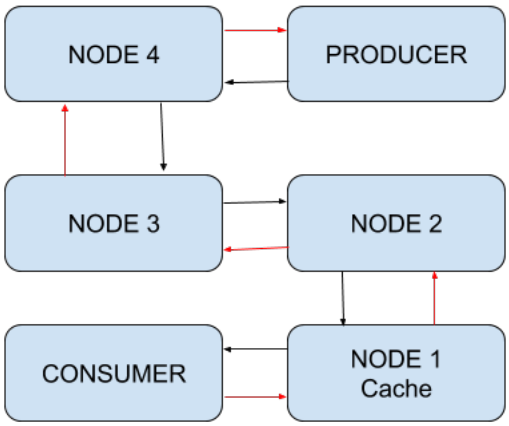

Figure 3.7: FATE packet in network

```
<Attribute name="Distance" nameType="4" dataType="5" data="0" />
```

Listing 3.4: Ingress Initial Packet Format

3.5 Life of A Packet

3.5.1 Life of a Packet: Internode Communication

It is desired to evaluate content based upon distance from producer. Farther content is more valuable. The consumer sends an interest packet (path marked in red). Producer receives an interest packet, and replies with a data packet (path marked in black). FATE uses a TLV format, so the packet may contain very few to many packet attribute fields. The producer adds a new packet attribute, distance, to the data packet in listing 3.4.

At each NODE, the data packet attribute 'Distance' is incremented (this is done by an evaluator similar to TTL) At Node 4, Distance is incremented to '1' At Node 3, Distance is incremented to '2' At Node 2, Distance is incremented to '3' At Node 1, Distance is incremented to '4' by a module, then evaluated by the cache module. The cache module takes the value '4' runs a normalizer function

```
<FATEPKT purpose="4" name="/fileNum=2" >
  <TempAttribute name="IPSRC" nameType="2" dataType="2"
      data="10.1.2.1" />
  <Other Layer3 (Temp) Attributes />
  <Original FATE packet Attributes />
</FATEPKT>
```

Listing 3.5: Initial IntraPacket Format

to rank it. Distances of less than 3 are less valuable, and distance greater than 4 are more valuable.

3.5.2 Life of a Packet: Intranode Communication

FATE uses a TLV packet, which supports two types of packet attributes: Native attributes, which always exist in the packet, and Temporary attributes, which only exist in the same Node. In NS3, FATE automatically registers new packet attributes, serialization and deserialization. No code changes! Native attributes are attributes which will exist outside a node. Attributes, such as a TTL field or Timestamps, are examples of this attribute. It may also be used for intranode communication (e.g., congestion may be relayed or hop count) Temporary attributes allow one module to pass information to other modules. This allows sharing of same evaluations for single/multiple modules. Examples: Security evaluation: Suspicious content should be less likely to be cached or forwarded than known safe content. Content Valuation (How valuable is the content): Cache may use this evaluation with other evaluators; such as Freshness, LRU (Least Recently Used), or HASH (hash value of a packet). Forwarding may use this evaluation to determine packet forwarding priority (against a packet buffer). Additional forwarding evaluators may include lifetime (e.g. expire packets over x milliseconds old), power cost to transmit a packet, or congestion on a specific path.

As shown in Figure 3.8 and represented in packet listing 3.5:

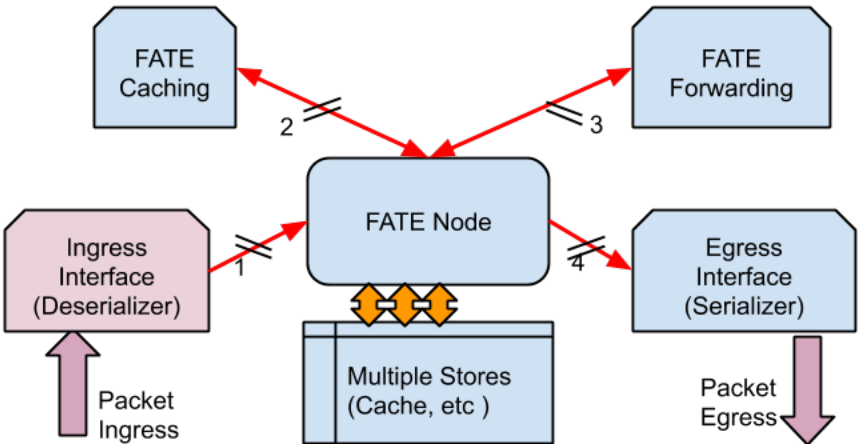

Figure 3.8: FATE packet ingress

Stage 1: Ipv4 packet is turned into a useable FATE format. Since NS3 sockets hide IPv4 information, we do not know the source destination in case of a cache hit. FATE includes the entirety of L3 information, such as source/destination IP addresses, ports, and other L2/L3/L4 field information.

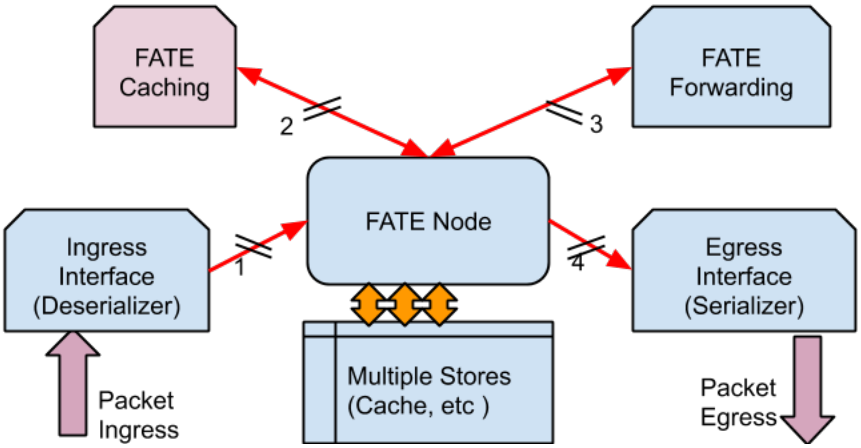

Figure 3.9: FATE packet internode information transfer

```
<FATEPKT purpose="4" name="/fileNum=2" >
  <TempAttribute name="CACHEHIT" nameType="3" dataType="4"
      data="1.0" />
  <TempAttribute name="IPSRC" nameType="2" dataType="2"
```

```
< FATEPKT purpose="8" name="/fileNum=2" >
  <TempAttribute name="CACHEHIT" nameType="3" dataType="4"
      data="1.0" />
  <TempAttribute name="IPDST: nameType="2" dataType="2"
      data="10.1.2.1" />
  <Attribute name="DATA" data="ff3e1411..." />
</FATEPKT>
```

Listing 3.7: Packet at forwarder, to determine forward interest packet or respond with data packet

```
      data="10.1.2.1" />
</FATEPKT>
```

Listing 3.6: Marking interest packet with cache hit

Stage 2 (Figure 3.9, Packet Listing 3.6): Assume this is a cache HIT. The Cache module marks (evaluates) the packet as a hit. This allows other modules to operate on the interest packet, and the forwarder to make changes. The cache tracks objects (internal eviction evaluation), and marks the packet as a cache hit.

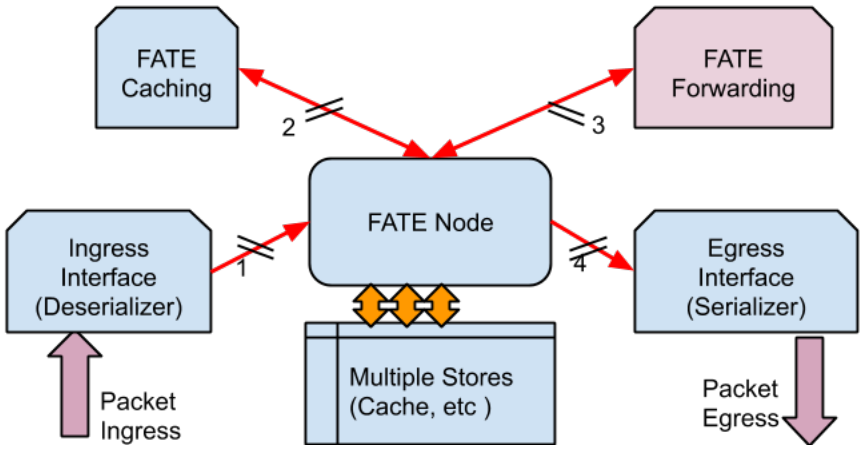

Figure 3.10: FATE packet internode information transfer: cache evaluation

Stage 3 (Figure 3.10, Packet Listing 3.6): FORWARDER evaluates the packets as a cache hit, based upon the 'CACHEHIT' field. The FORWARDER changes the packet type from interest to data, adds the correct data attribute and reverses the IP SRC/DST fields. If the packet was a miss, it would forward the packet to the next node towards the destination.

```
<FATEPKT purpose="8" name="/fileNum=2" >
  <TempAttribute name="CACHEHIT" nameType="3" dataType="4"
      data="1.0" />
  <TempAttribute name="IPDST: nameType="2" dataType="2"
      data="10.1.2.1" />
  <Attribute name="DATA" data="ff3e1411..." />
</FATEPKT>
```

Listing 3.8: Data packet to be egress

Stage 4 (Figure 3.11, Packet Listing 3.7): FORWARDER evaluates the packets as a cache hit, based upon the 'CACHEHIT' field. The FORWARDER changes the packet type from interest to data, adds the correct data attribute and reverses the IP SRC/DST fields.

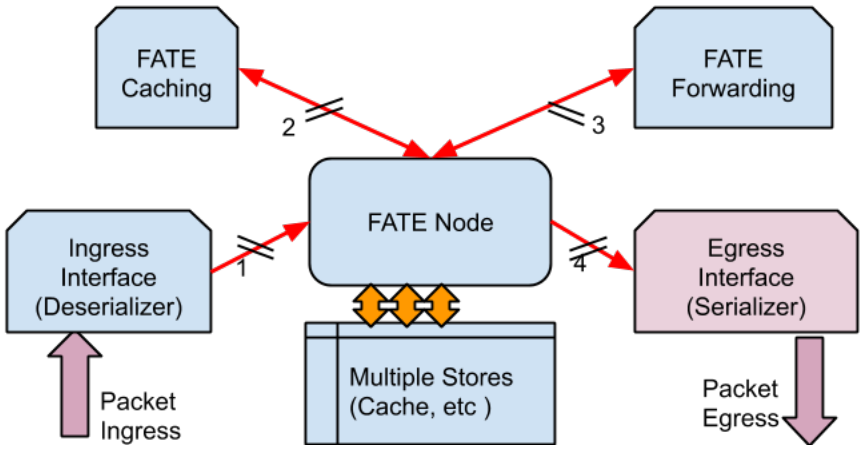

Figure 3.11: FATE packet internode information transfer : forward evaluation

Stage 5 (Figure 3.12, Packet Listing 3.8): FATE to IPv4 Packet, using the TempAttribute fields to recreate the necessary layer2/3/4 fields to transmit the packet. The data field of L4 (UDP) carries the serialized FATE packet. Before this field is added, all temporary attributes are removed. Valid IPv4 data packet is transmitted to IP 10.1.2.1 (the original source IP for the interest packet).

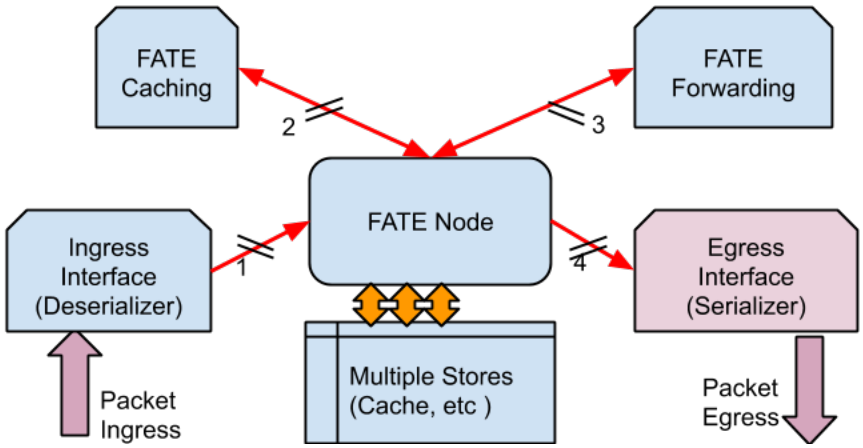

Figure 3.12: FATE packet egress

3.6 Modules

FATE, itself, is composed of several modules. The central or root node is called a 'Node'. The Node may have several other modules assigned to it. At a minimum, a Node has a Forwarding Module, but may have many more. Each module has a specific function. A module has unique logic and actions, based upon evaluations. Each module is charged with tracking, storing, or removing Information based upon any internal algorithms. Typically, Each module has a single Combinational Function Utility assigned to it. This is, simply, a combinational logical organization of the atomic evaluators, typically connected together to be called a Utility. Each module has a unique name assigned to it, which is used for control packets, configuration, and logging functions (described below). Every module has a single evaluation method for packets, but the purpose of each module is up to individual design. Some nodes may have a cache module, while others may have a cache and prefetch module.

Figure 3.13 shows a traditional architectural model. Each module, independently, processes packets or creates new packets. Discovery module will send out

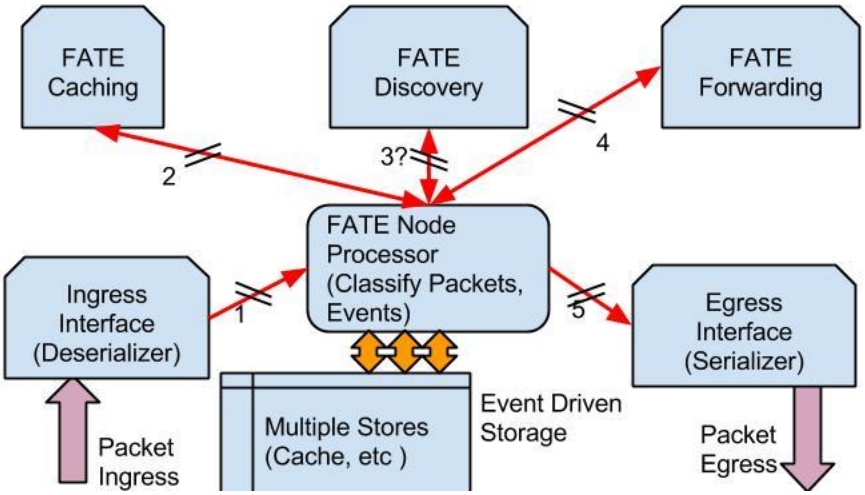

Figure 3.13: Traditional FATE node setup

and process Discovery packets (e.g. Hello packets), updating results in a store. The Forwarding module will use the same table, in conjunction with a decision from a Utility Forwarding block, to decide if, and where, the packet should be forwarded in the Egress Interface.

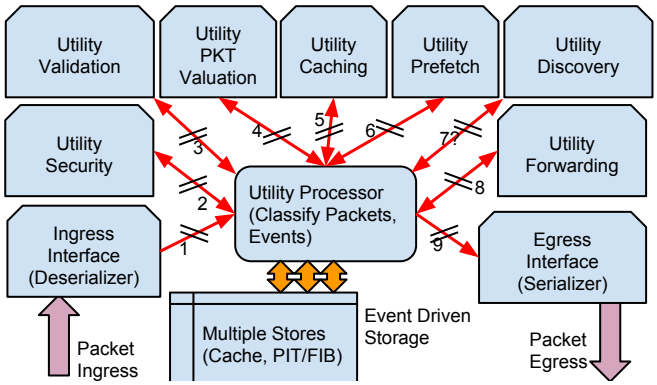

Figure 3.14: Example of an expanded FATE node setup

Figure 3.14 shows a variant node, with additional modules. This expanded capability node shows how easy it is to add features to FATE. In this example, packets are processed and evaluated in Utility Security (with a temporary meta-data tag, to gauge authenticity of the packet) and Utility Validation (to measure

how valid is the packet). A packet corrupted by noise can be measured here, using the ingress port and (if uncorrupted) L2 next neighbor address. It is useful for storing the noisiness of an interface in a table (to allow Forwarding Utility to avoid higher bit-rate error connections).

Next, the packet is given a temporary meta-data result tag from valuation (to be used in caching and forwarding), which is the evaluation of the importance of the packet (e.g. TOS or COS features). Utility Caching will evaluate Interest Packets on their cacheability, and store valid Data packets. Utility Prefetch will measure the likelihood of a related packet being requested soon (such as the next 'n' sequences of a file, or prefetch files in a manifest).

Utility Discovery will attempt to find new neighbors and process results from other nodes. Utility Forwarding, using the prior temporary meta-data tags from Utility Security, Utility PKT Evaluation to egress the packet out a port (after being evaluated against each egress port, with respect to the BER table, updated by the Utility Validation module). As shown in Figures 3.13 and 3.14, each module performs a specific function, either to update stores (tables), in support of evaluating in other modules (Utility Security, Utility PKT Evaluation), or perform an action (Utility Caching/Prefetch/Forwarding).

3.6.1 Statistics

Statistics are handled in a very different way than most simulators. Statistics are logged by name-value pairs. The name is node_name/module_name/statistic_name hierarchical format, and the value can be any value, typically integer or double. This allows statistics to be retrieved or updated, by node name, module, statistical name, or any combination thereof. Since FATE is overly flexible, and new modules can be created, it is up to each unique module to define, specific to itself,

which statistics are used, and when/where to update them.
The following methods are support in statistics: GetStats, SetStats, IncrementStats, DumpStats.

3.6.2 Logging

Logging is initialized from the configuration file, and is set up on a per node basis, with the exemption of a simulator, which has the option of individual or a single global logging function. FATE logging, like statistics, can be done at the node, module, or event level. For the event level, it is similar to most logging schemes, such as ALWAYS, DEBUG, WARNING, and other criteria to log. FATE can use the native logging capabilities of a simulator (such as NS-3), but it is the responsibility of the logger itself to convert between FATE and simulator specific events, for logging purposes. Internally, FATE uses a similar output stream as C++ redirection (e.g. std::out command).

3.6.3 Node Overview

The Node is the container of modules, with each module having a specific job.

3.6.4 Asynchronous Events

FATE supports asynchronous events, including timer expirations (which are set per module), or node events (e.g. tx fifo empty or packet received). This allows watchdog timers, if desired, to remove stale store contents, or create necessary discovery packets.

FATE can be imagined as a tree (Figure 3.15, 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

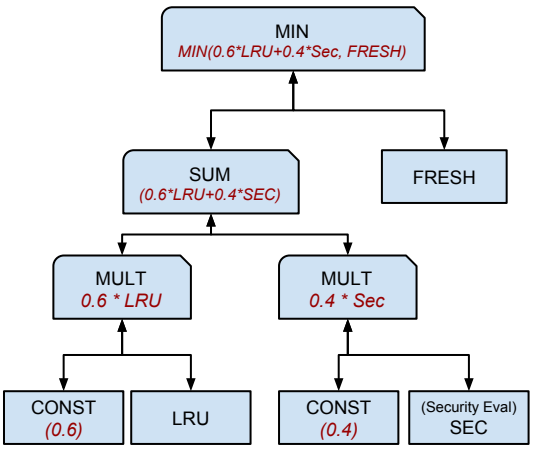

Figure 3.15: Utility caching tree representation of MIN(.6*LRU+.4*SEC, FRESH)

aggregated branches, towards the root, to give an evaluated result. As a simple example, imagine a system which desires content to be distributed throughout a system (either a cache prefetch, or a MANET which shares commonly used content), but is subject to cache poisoning. To easily model this, we need two main (atomic) functions: One to authenticate content (to avoid cache poisoning) and the other is a traditional caching algorithm (LRU). In Figure 3.15, shows a cache which is weighted 60% LRU, and 40% SECURITY (a value from a prior module, evaluating the authenticity of the content). FRESH represents if the content is not stale (no need to cache obsolete or stale content). This can be represented (as shown by the tree) or by the algebraic formula *MIN(.6*LRU + .4*SEC, FRESH)*.

3.6.5 Caching Module

FATE, in contrast, uses an algebraic formula to evaluate *each* piece of information in the module's local store, and evicts the content with the lowest value. Depending on the functions used, this can is a trade off between flexibility and

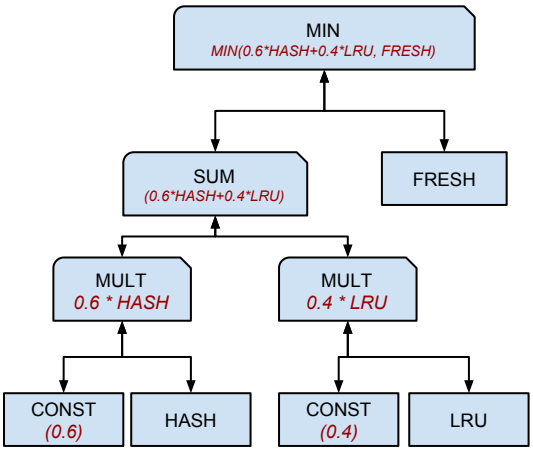

Figure 3.16: Utility caching tree using HASH of an NDO attribute

rapid development, against speed of simulation. A slight modification for FATE, enables caching on a hash of the content's name, data itself, or any attribute, as shown in Figure 3.16. This allows on-path caching, where each node on the path will (mostly) cache name or content matching the hash declaration (e.g. *HASH(data)* modulus N = [x,y] is true). Since the utility cache configuration is done via XML, no recompiling or formulating new code for simple changes. Only if a new module or new function is necessary, will compilation be necessary. Algebraic representation of Information evaluation greatly increases new algorithmic development, as it is only necessary to identify which functional evaluations will be done, and how they will be weighted.

3.6.6 Forwarding Module

Forwarding is accomplished by evaluating each tuple of Information with the available egress ports and 1-hop neighbors. The highest evaluation, per egress port, is chosen, using the corresponding 1-hop neighbor address. Evaluation, per port, is configurable, and typically is an evaluation of the port characteristics combined with a NDO evaluation with the next-hop neighbor evaluation.

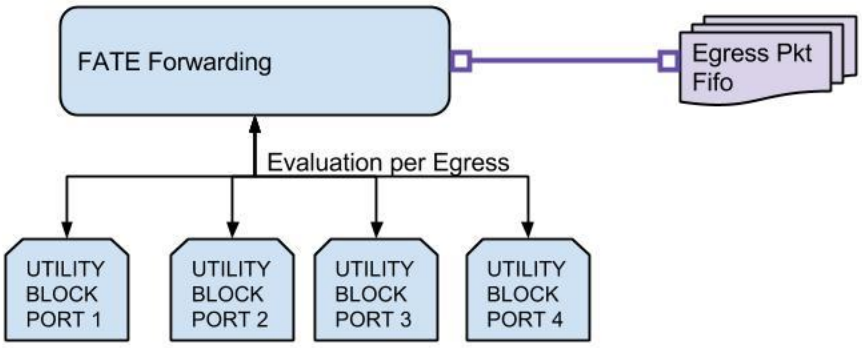

Figure 3.17: Example forwarding utility evaluation, per egress port, with a pending Packet FIFO

Each egress port is evaluated for suitability of packets (figure 3.17). Port evaluations, typically, include (if it has) collision detection, to return zero, until a specified timer expires. Bit error rate, per port, can be configured between [0,1], depending on the noise of the line.

Packet Evaluation can be done by TOS/COS, to give certain QoS packets higher priority, or even packet size (easier to send smaller packets than single large packet). While the next neighbor evaluation (per egress port) can be done by Hops, latency, and/or bandwidth.

Figure 3.18 shows the evaluation, from a physical implementation view point. Node 1 has two ways to communicate with Node 2 (bluetooth and wifi), while the wifi physical port has two neighbors (Node 2 and 3). In addition, Node 4 and 5 have separate wired connections. Each outgoing packet is evaluated against these criteria. A packet, outgoing to Node 2 may use either bluetooth or wifi, but another packet may only use wifi (Node 3). Fate has an option to evaluate outgoing content by block ranges. As an example, content evaluated below 0.2 may not be sent (a better egress path may not be immediately available), while content between between [0.2,0.6] may be considered the same, and subject to a

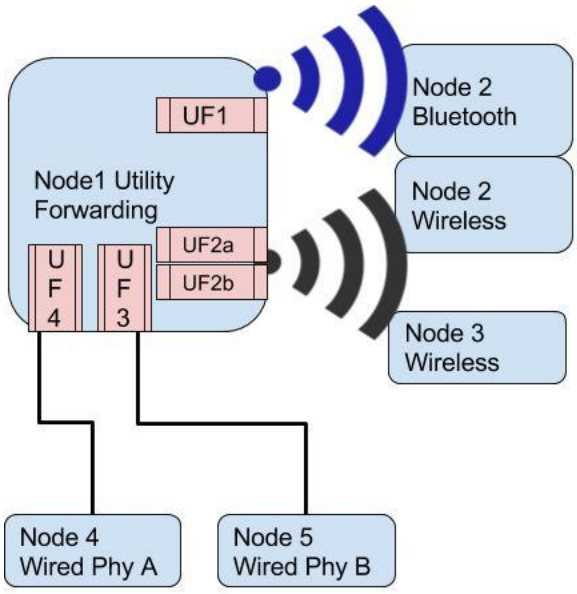

Figure 3.18: Example utility forwarding evaluation per PHY and connection

knapsack solution.

The forwarding module can be implemented in many different ways. It can be implemented to create a custom forwarding scheme, or use an existing protocol, and evaluate the forwarding methodology for it. Both native (soley by Utility evaluation) and protocol (using a known protocol, such as IPv4 and UDP/TCP) are supported.

3.6.7 Stores

Stores are the globally persistent repositories. Content Stores will cache Data packets, save persistent data, such as link-state information for forwarding, security information, bit-error rates, and other needs.

3.7 Licensing

Fate uses pugixml([54]), which is authored under the freebsd license. Pugixml is a simple, easy to use, and convenient implementation of XML, which is used in configuration. FATE ([68]) itself uses a modified freebsd license, which allows the code to be as desired, with the exception of crediting the source, or permission from the owner.

3.8 Intermediate-Directed Forwarding

FATE can also be used to conditionally control intermediate routing points. FATE can have a chain of intermediate points, allowing the routing protocol to route between specific points. The Figure 3.19 shows a traditional IP/ICN path. The same path is followed, to/from the consumer and producer. The protocol is limited by its own algorithmic restrictions on what is a best path. If part of the path is suspected of having a man-in-the-middle, intercepting information between end points, or if a section of the network is known to be having intermittent problems, it can be bypassed with FATE. As shown in Figure 3.20, the traditional path is bypassed. FATE specifies a specific intermediate gateway, from which the final destination is used. In fact, if more security is desired, by having the ingress/egress path from the consumer to the producer to be different, FATE easily allows that accomodation. In Figure 3.21, both the request packet path, and the response packet path, take different intermediate node points. This allows denying a man-in-the-middle both ends of the communication path. Normally, HTTP allows a single intermediate node (TCP packet source, destination (CDN), and the HTTP 'Host:' header, for the actual sourced content. This allows HTTP to have end point connection with a CDN, but retain the expected/original end

point if content is missing from the CDN cache. While FATE redirect routing is similar, it has no limitations to the number of intermediate nodes. This allows a greater control, from the source, to direct to the correct cache, a specific gateway, or alternate route (which would not be possible with IPv4 routing).

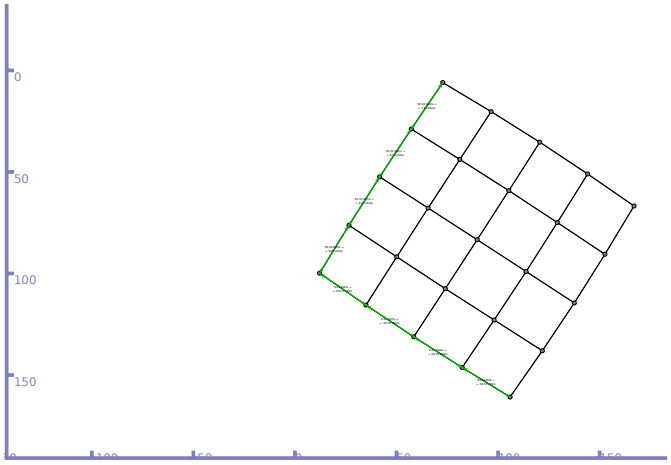

Figure 3.19: Traditional routed traffic

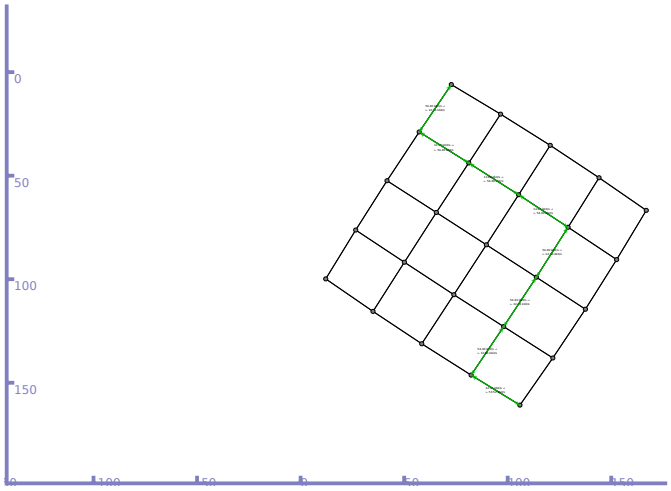

Figure 3.20: Regular directed unidirectional traffic

As shown in Figure 3.19, the traditional IPv4 path is chosen. In Figure 3.20, FATE sets an intermediate node point at node 5, then to node 8, node 23 and the final destination of node 24. The return path, in this instance, is chosen to be the

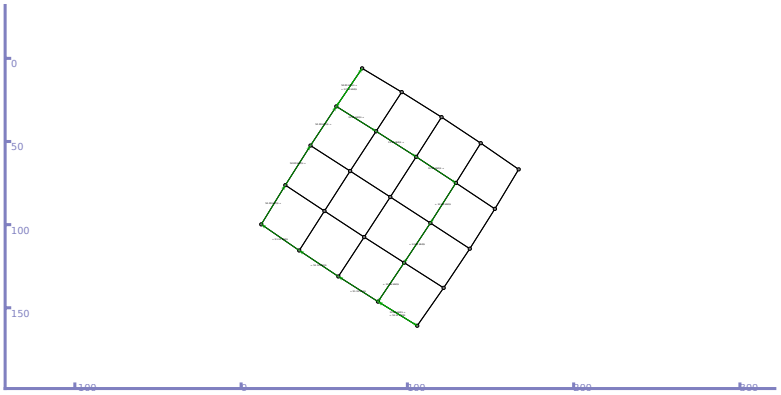

Figure 3.21: Dual egress-ingress routed traffic

same. In this instance, it may be useful to have an alternate path chosen, which does not reflect IPv4 shortest destination criteria. The nodes may reflect a higher security connection, suspect content which needs additional on path analysis, or other criteria. The producer will use the same path, in reverse order, to return to the originating source node. Figure 3.21 shows both the ingress and egress paths may be different. This is extremely useful against man-in-the-middle attacks, as the attacker can only see either the requests or the responses, but not both, due to the packet path variation. The return path is set by the producer. In all cases, if desired content is found in an intermediate caching node, the content is returned, without progressing the entire network.

FATE allows intermediate routing, where the traditional routing algorithm is used between each end-point, until the destination is reached. Each route taken by the interest, data or other packet types may have a unique pattern. Controlling intermediate destinations allows other options, such as hashed-caching, or centralized locations for pub-sub networks.

Chapter 4

Flexible Evaluation Caching Using FATE

4.1 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 [25]), which uses omnet++ [101] 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 reimplementation on another platform, or a reimplementation of the competing algorithm, which may be prone

to interpretation or other errors.

FATE can minimize or negate these problems by allowing its modular nature to enable quick evaluation of potential algorithms, and use the algebraic expression to, consistently, be used for evaluation and comparison to other algorithms.

4.2 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 [78] 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 [63] 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 [64], 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 [31] 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 a significantly faster time frame. FATE does not replace a network simulator, but offers an alternative Node (switch) implementation. NDN [11], [67] 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).

4.3 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, or Linux), 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 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.

4.3.1 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.

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) returns the minimum value of its inputs.

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

ADD : ADDITION(a,b,..,z) returns the sum of all its input. If the sum is greater than 1.0, it will require scaling.

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

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

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.

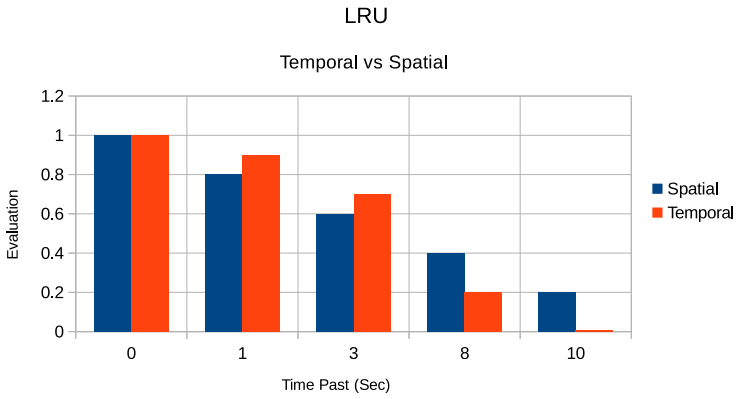

Figure 4.1: Temporal vs spatial LRU values

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 4.1). 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.

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 4.2). 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.

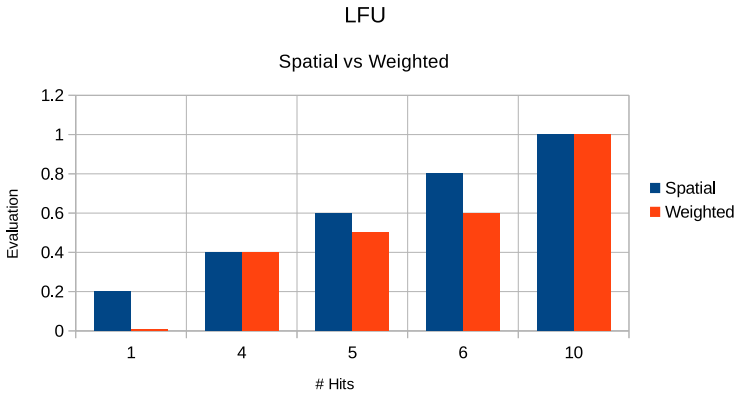

Figure 4.2: Weighted vs spatial LFU values

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 orderings 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).

4.3.2 Modules

The structure of FATE can be viewed as a tree, as illustrated in Figure 4.3, with atomic functions being the leaf, and algebraic aggregate functions being branches. The information packet is passed down and evaluated at each leaf.

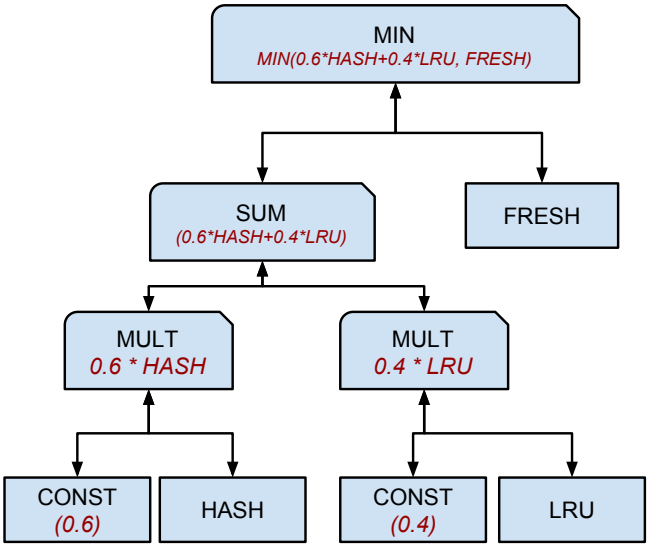

Figure 4.3: Utility caching tree representation of MIN(.6*HASH+.4*LRU, FRESH)

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 be 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.

4.4 Results

FATE caching was evaluated using two simple network scenarios. Figure 4.4 illustrates the first scenario of a simple network with a single caching node between a client and a server. Figure 4.5 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.

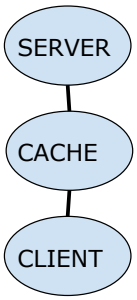

Figure 4.4: Simple single caching node

There is no consensus on what the correct Internet traffic distribution is, with some support for a Mandelbrot distribution [48], while others support a Zipf distribution [19]. 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 4.1 uses the default adopted in NDNSim with a 'q' setting of '0.7' for the Mandelbrot distribution. Table 4.2 shows the results of the Mandelbrot

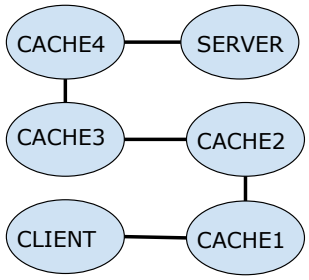

Figure 4.5: Four Caching Node Network

'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 4.1: 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%

The results show that the atomic FATE LRU evaluator ranked values similarly to the traditional algorithmic LRU implementation in NDN. LFU significantly

Table 4.2: 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%

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 value declines over time.

According to results published by Dabirmoghaddam et al.[32], 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 intermediate results of the two.

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

Table 4.3: 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

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+LRU*0.8)*LFU*HASH. As shown by the hit rate per individual node, 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: MIN((0.2+LRU*0.8)*LFU, 0.2+0.8*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 to 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.

Chapter 5

Extended Caching

5.1 Introduction

Caches have finite capacity. This affects not only how much content can be available for cache hits, but affects the efficiency of certain cache eviction algorithms. For these algorithms, such as LFU, the state can be changing, which is not reflective of future demand. If LFU only could cache 5 large objects, it would become difficult to reflect the most requested content, based upon a small counting index. What is desired is an extended counting index, beyond the actual capacity of the cache. The extended index is an ethereal state, reflecting more objects for better caching, but having the same content capacity as the original, smaller cache. LRU does not, natively, have an ethereal state. If an object is evicted, then requested, the prior weighted value is lost; only algorithms, such as LFU, which require a prior state (or count) can have an ethereal state.

In this experiment, a cache size of 'x' (10 and 50 content entries) are available, but FATE is stateful, which allows us to keep an extended ethereal state for unavailable values (50 and 2500 ethereal entries, respectively). Since content consumes significantly more memory than an ethereal entry (which is an integer),

it produces better results for the given cache size, but not as expensive as having a cache size matching the ethereal size.

As an example, 10/50 has a physical cache size of 10 entries, but records the ethereal values for up to 50 objects.

5.2 Results

The results of LFU and the extended-LFU are in the table 5.1. Ethereal refers to the quantity of tracked named-content, while the cache value refers to the actual (maximum) number of cacheable objects stored. The results show an improvement from the physical cache size, but not as significant as the ethereal size. This feature allows greater cache-ability at the expense of a few bytes of memory.

Using the extended values in table 5.1, we present the cache-ethereal values for 10C-E50, 50C-E2500, and the fixed LFU values for cache sizes of 10,50,and 2500. To show the actual effectiveness of extended-LFU, comparison values of real to ethereal are shown: The increase cache rate of 10C-50E is compared to LFU fixed of cache size 10 (and ethereal size 10), and decrease from cache size of 50. Likewise, we compare 50C-2500E to LFU cache sizes of 50 and 2500.

5.3 Conclusion

The results show an intermediate hit rate from Extended FATE caching (small cache, large ethereal state tracking) from a traditional small and large cache. Comparison in table 5.2 and table 5.1, shows the improvement and the conclusion that extending the ethereal state, without increasing the content store, does enable a higher cache hit rate. For stateful caching algorithms, using an extended, ethereal

Table 5.1: LFU cache size x / (ethereal) record Y entries

algorithm zipf	run1	run2	run3	run4	run5	avg	std dev
LFU 10/10							
0.5	0.03%	0.21%	0.20%	0.30%	0.09%	0.17%	0.10%
0.8	5.43%	5.90%	5.75%	5.90%	6.44%	5.89%	0.37%
1	21.73%	22.06%	21.04%	20.33%	21.82%	21.40%	0.71%
1.2	45.41%	42.37%	41.42%	46.88%	44.59%	44.13%	2.23%
0.5	0.02%	0.01%	0.01%	0.01%	0.01%	0.01%	0.01%
0.8	0.02%	0.39%	0.07%	0.47%	0.13%	0.22%	0.20%
1	1.91%	10.10%	2.27%	3.13%	2.70%	4.02%	3.43%
1.2	35.67%	36.70%	11.90%	29.32%	24.60%	27.64%	10.08%
LFU 50/50							
0.5	0.75%	0.77%	0.71%	0.77%	0.56%	0.71%	0.09%
0.8	11.23%	11.83%	10.86%	11.56%	11.53%	11.40%	0.37%
1	33.30%	33.14%	33.27%	33.35%	33.81%	33.38%	0.25%
1.2	61.09%	61.02%	61.05%	61.06%	60.96%	61.04%	0.05%
LFU 2500/2500							
0.5	8.68%	8.65	8.71	8.68	6.23	8.48%	-
0.8	35.32%	35.42%	35.30%	35.56%	35.32%	35.38%	0.11%
1	64.13%	63.79%	64.02%	64.21%	63.68%	63.97%	0.23%
1.2	85.70%	85.83%	85.93%	86.07%	85.93%	85.89%	0.14%
LFU 10/50							
0.5	0.49%	0.53%	0.60%	0.57%	0.30%	0.50%	0.12%
0.8	7.67%	7.73%	7.75%	7.80%	7.53%	7.70%	0.10%
1.0	26.00%	25.97%	25.89%	26.04%	26.04%	25.99%	0.06%
1.2	55.35%	55.15%	55.40%	55.08%	55.69%	55.33%	0.24%
LFU 50/2500							
0.5	1.91%	1.90%	1.90%	1.92%	1.92%	1.91%	0.01%
0.8	17.53%	17.80%	17.66%	17.64%	17.52%	17.63%	0.11%
1.0	50.36%	50.22%	50.30%	50.56%	50.11%	50.31%	0.17%
1.2	81.78%	81.89%	81.97%	82.22%	81.89%	81.95%	0.16%

Table 5.2: Hit rate improvement of extended ethereal vs traditional caching

C10-E50	Zipf	Ethereal	Fixed	Increase	Decrease
		LFU	LFU	from C10	from C50
	0.5	0.50%	0.17%	194.12%	-29.58%
	0.8	7.70%	5.89%	30.73%	-32.46%
	1.0	25.99%	21.40%	21.45%	-22.14%
	1.2	55.33%	44.13%	25.38%	-9.35%
C50-E2500	Zipf	Ethereal	Fixed	Increase	Decrease
		LFU	LFU	from C50	from C2500
	0.5	1.91%	0.71%	169.01%	-78.00%
	0.8	17.63%	11.40%	54.65%	-50.17%
	1.0	50.31%	33.38%	50.72%	-21.35%
	1.2	81.95%	61.04%	34.26%	-4.59%

state helps cache efficiency.

Chapter 6

QoS Caching

6.1 Introduction

Internet content uses a well known distribution, which is mandelbrot-zipf in nature. Many internet caching eviction protocols, such as LRU and LFU, take advantage of this distribution to maximize caching efficiency.

But, other types of networks may not have a zipf-ian distribution. Other networks may have other criteria, or constraints, for caching. These constraints may be dictated by small cache size [21], optimization for video playback [113], pocket or local networks [77] [24] [50], actual type of content/service such as video [70], availability of content affected by long distances to the producer [81] [79], or having a packet marked to dictate content importance (similar to the TOS flag on tcp). Each criteria can not only be unique, but have several additional criteria to dictate content importance.

FATE, which uses evaluation to dictate content value, as opposed to a strict algorithmic formula, can be used for QoS caching. To illustrate this, we will create a requirement for cascading constraints to be issued, requiring the prior results for each stage to be done.

For the following examples, three consumers will request different content, from three different producers. Each producer is set at a different distance from the consumers. In addition, each producer will be ranked differently for preferred caching, and which requested content may contain important messages.

In each case, the XML configuration changes are shown, as well as the cache state, at the time of completion. First, each individual evaluation will be used: LRU, SIZE, QoS (field value dictating preferred customer content value), DISTANCE, and REGEX (matching specific field value) are used to see the effects of each individual evaluator. In the formula, a small randomness and the last packet inserted are protected, to allow a more correct eviction of content, when necessary.

After each individual QOS evaluator is used, the requirements become complex and are combined into an expression, to choose the best evaluation of LRU, SIZE, and DISTANCE, unless a QOS field is set or a REGEX matching pattern in the content.

1. LRU - First LRU is used to show results of repeatability.

2. SIZE - The size of the content is considered for eviction. Smaller content is higher ranked, than larger content.

3. QOS - Traditional field used to signify customer and value. Similar to TOS/-COS fields.

4. REGEX - Allows regex matching in a packet field, used to indicate high priority fields.

5. DISTANCE - Producers which are farther away are given higher rank than closer producers. It is more expensive in network lag to retrieve content farther away.

6. LRU*SIZE - The first complex QOS cache algorithm, considering size and content request frequency.

7. MAX(LRU*SIZE,QOS) - Similar to above, but we consider size and content, against customer ranking.

8. MAX(LRU*SIZE,QOS,REGEX) - Above, plus checking for URGENT fields.

9. MAX(LRU*SIZE*DISTANCE,QOS,REGEX) - Above, but consider distance, size, and content frequency to urgent and customer fields.

6.2 Results

Results are presented for each individual and combination of QOS caching. Each section will show the relevant XML configuration segment, and the cache contents. Cache hit rate is an effective metric for comparison of caching algorithms which are comparing over zipf distribution (frequently requested content). By showing a cache dump, the desired effects can be seen, as applied for each specific QOS requirement.

Three producer nodes (and clients) are used to show the effect of distance (the content is labelled /test1, /test2, and /test3).

6.2.1 FATE LRU effects

Starting with standard LRU and the complete module configurations for LRU.

SIZE XML configuration snippet

A full XML configuration file is presented in listing 6.1 . LRU is chosen, using the 'ceiling' normalizer.

```
<NodeModule>
  <UtilityModule moduleName="CacheBasicManager"
    associatedStore="CacheStore" cacheSize="370"
      ContentTypes="File" > <!--File" or "Icn"-->
    <Utility name="UTILITYBLOCK" order="1" >
      <Utility name="LRU" >
       <Normalize normalizeName="NormalRanked"
        value_type="ceiling" />
      </Utility>
    </Utility>
  </UtilityModule>

  <UtilityModule moduleName="SecurityBasicManager" >
    <Utility name="UTILITYBLOCK" proxyName="SecurityBlock" order="1">
      <Utility name="COUNT" missing_count_value="0"
        matching_lower_bound="0" matching_upper_bound="1"
         count_condition="increment"
        match_criteria="LeftRightInclusive" attribName="Distance" />
    </Utility>
  </UtilityModule>

  <UtilityModule moduleName="ForwardNs3Ipv4Manager3" order="3"
    associatedCacheStore="CacheStore" >

    <Utility name="UTILITYBLOCK" order="1">
        <Utility name="NAMECHAIN" appendNodeName="true"
        defaultAttribute="Path:" nodeNamePartition=" ; "
        appendIfnExist="true" appendInFront="false"
        nodeNameUnique="false"/>
    </Utility>
  </UtilityModule>
  <Store storeName="CacheBasicStore" name="CacheStore" size="370"
    storageMethod="MemMap"/>
</NodeModule>
```

Listing 6.1: Base Packet XML format

Cache Snippet of LRU with alpha=0.8

Below are the zipf alpha used, and a dump of the cache upon completion. Traditional caching uses hit rates, to reflect availability of most requested content. But in QoS caching, the desired content may not be the more requested content. Thus, to help show the effects of QoS caching, and which elements were purged due to low valuation.

LRU a0.8

```
CACHED: /test1/fileNum=5 = 0.381665
CACHED: /test1/fileNum=3 = 0.382289
CACHED: /test1/fileNum=79 = 0.382351
CACHED: /test2/fileNum=1 = 0.383386
CACHED: /test3/fileNum=1 = 0.384688
CACHED: /test3/fileNum=34 = 0.385511
CACHED: /test3/fileNum=12 = 0.38626
CACHED: /test3/fileNum=6 = 0.387108
CACHED: /test3/fileNum=16 = 0.393201
CACHED: /test1/fileNum=32 = 0.395977
CACHED: /test1/fileNum=10 = 0.398969
CACHED: /test1/fileNum=50 = 0.39917
CACHED: /test3/fileNum=67 = 0.39983
CACHED: /test3/fileNum=3 = 0.580155
CACHED: /test2/fileNum=2 = 0.782019
CACHED: /test1/fileNum=1 = 0.981504
PURGE: Store size is 400/370,
        erase /test1/fileNum=5 value of 0.381665
PURGE: Store size is 380/370,
        erase /test1/fileNum=3 value of 0.382289
```

Cache Snippet of LRU with alpha=1.2

Using LRU with the request rate set to $alpha = 1.2$, a snippet of cache is below.

LRU a1.2

```
CACHED: /test1/fileNum=19 = 1.92868e-311
CACHED: /test3/fileNum=49 = 0.0559752
CACHED: /test1/fileNum=46 = 0.111284
CACHED: /test3/fileNum=78 = 0.167037
CACHED: /test1/fileNum=32 = 0.222346
CACHED: /test3/fileNum=34 = 0.278099
CACHED: /test1/fileNum=5 = 0.333407
CACHED: /test2/fileNum=1 = 0.388716
CACHED: /test3/fileNum=1 = 0.388716
CACHED: /test1/fileNum=4 = 0.5
CACHED: /test3/fileNum=12 = 0.666815
CACHED: /test2/fileNum=2 = 0.667037
CACHED: /test3/fileNum=67 = 0.777877
CACHED: /test1/fileNum=1 = 0.833185
CACHED: /test3/fileNum=16 = 0.888938
CACHED: /test2/fileNum=4 = 0.88916
CACHED: /test3/fileNum=6 = 1
PURGE: Store size is 430/370,
        erase /test1/fileNum=19 value of 1.92868e-311
PURGE: Store size is 420/370,
        erase /test3/fileNum=49 value of 0.0559752
PURGE: Store size is 410/370,
        erase /test1/fileNum=46 value of 0.111284
PURGE: Store size is 400/370,
        erase /test3/fileNum=78 value of 0.167037
PURGE: Store size is 390/370,
        erase /test1/fileNum=32 value of 0.222346
PURGE: Store size is 380/370,
        erase /test3/fileNum=34 value of 0.278099
PURGE: Store size is 370/370,
        erase /test1/fileNum=5 value of 0.333407
```

6.2.2 FATE SIZE effect upon caching

File definition for content sizes

The next block deals with object size. To mimic different object sizes, a file contains a description of the sizes. The first entry gives the block size (10 bytes). Each block is requested per request, thus requesting four blocks of 10 bytes will result in 4 packet requests of 10 bytes each. The next entry maps to the first request (fileNum=1 of size 4 blocks, each block/packet request is 10 bytes, for content size of 40 bytes), the next entry maps to the second request (fileNum=2), etc. The last entry represents the nth request and all subsequent requests. The file representing the content sizes used is below:

```
<UtilityModule moduleName="CacheBasicManager" order="2"
    associatedStore="CacheStore" cacheSize="370" ContentTypes="File">
  <!--File" or "Icn"-->
  <Utility name="UTILITYBLOCK" order="1">
    <Utility name="MAX">
      <Utility name="SUM">
        <Utility name="MULT">
          <Utility name="RawEval" attribName="TotalSize">
            <Normalize normalizeName="GeometricMatch" biasLowVal="false"
                invertValue="false" divisor="10"/>
          </Utility>
          <Utility name="CONST" defaultValue="0.92"/>
        </Utility>
        <Utility name="MULT">
          <Utility name="CONST" defaultValue="0.02"/>
          <Utility name="RND" randomType="alwaysRnd"/>
        </Utility>
      </Utility>
      <Utility name="PLE"/>
    </Utility>
  </Utility>
</UtilityModule>
```

Listing 6.2: Size Eval XML format

```
10    //size of each block in bytes
4     //size of 1st element (40 bytes)
3     //size of 2nd element (30 bytes)
4     //size of 3rd element (40 bytes)
5     //size of 4th element (50 bytes)
2     //size of 5th element (20 bytes)
7     //size of 6th element (70 bytes)
1    //size of 7th (and beyond) file size of 10 bytes
```

SIZE XML configuration snippet

The XML segment for size is shown in listing 6.2:

Cache Snippet of SIZE with alpha=0.8

With a consumer request rate of $alpha = 0.8$, cache dump at the end (as there is a bias towards smaller files, most of the content is small). Request rate has no bearing on size functionality, but is shown as later complex formulation uses LRU in the context. From the configuration file, listing 6.2, shows the exact formulation

as MAX($SIZE*.92+RND*0.02$, PLE). This takes the size evaluation (block size '1' is 1.0, block size '2' is 1/2, size 'n' is 1/n ...). The size evaluation is multiplied by a constant value of 0.92, then a random number ([0,0.02]) is added. To protect the latest/newest content, it is maximized with PLE (protect last element ; which the last inserted element returns a 1.0).

SIZE a0.8

```
CACHED: /test3/fileNum=6 = 0.134241
CACHED: /test1/fileNum=17 = 0.920049
CACHED: /test3/fileNum=21 = 0.920733
CACHED: /test2/fileNum=92 = 0.92084
CACHED: /test1/fileNum=73 = 0.922281
CACHED: /test1/fileNum=13 = 0.92387
CACHED: /test3/fileNum=68 = 0.923917
CACHED: /test1/fileNum=43 = 0.923955
CACHED: /test3/fileNum=78 = 0.924171
CACHED: /test3/fileNum=49 = 0.924173
CACHED: /test2/fileNum=21 = 0.924611
CACHED: /test3/fileNum=15 = 0.925001
CACHED: /test3/fileNum=13 = 0.925281
CACHED: /test2/fileNum=8 = 0.925875
CACHED: /test3/fileNum=17 = 0.926299
CACHED: /test2/fileNum=66 = 0.926728
CACHED: /test2/fileNum=10 = 0.927455
CACHED: /test1/fileNum=19 = 0.92753
CACHED: /test3/fileNum=16 = 0.929747
CACHED: /test2/fileNum=34 = 0.930145
CACHED: /test1/fileNum=32 = 0.930224
CACHED: /test1/fileNum=50 = 0.931866
CACHED: /test2/fileNum=7 = 0.932153
CACHED: /test2/fileNum=100 = 0.932904
CACHED: /test3/fileNum=67 = 0.934209
CACHED: /test3/fileNum=29 = 0.935101
CACHED: /test3/fileNum=12 = 0.935752
CACHED: /test3/fileNum=60 = 0.938138
CACHED: /test1/fileNum=46 = 0.939302
CACHED: /test1/fileNum=56 = 0.939484
CACHED: /test2/fileNum=55 = 0.939774
CACHED: /test1/fileNum=10 = 1
PURGE: Store size is 380/370,
        erase /test3/fileNum=6 value of 0.134241
```

Cache Snippet of SIZE with alpha=1.2

$alpha - 1.2$ dump at the end (as there is a bias towards smaller files, most of the content is small), except the last inserted element.

SIZE a1.2

```
CACHED: /test1/fileNum=1 = 0.242089
CACHED: /test3/fileNum=5 = 0.467078
CACHED: /test3/fileNum=10 = 0.920831
CACHED: /test1/fileNum=29 = 0.921264
CACHED: /test2/fileNum=7 = 0.922507
CACHED: /test1/fileNum=10 = 0.922754
CACHED: /test1/fileNum=15 = 0.923437
CACHED: /test2/fileNum=24 = 0.924349
CACHED: /test1/fileNum=90 = 0.924506
CACHED: /test2/fileNum=60 = 0.925119
CACHED: /test3/fileNum=11 = 0.926779
CACHED: /test1/fileNum=14 = 0.92702
CACHED: /test1/fileNum=34 = 0.927109
CACHED: /test3/fileNum=43 = 0.927306
CACHED: /test3/fileNum=34 = 0.927844
CACHED: /test3/fileNum=7 = 0.929535
CACHED: /test2/fileNum=53 = 0.929796
CACHED: /test3/fileNum=8 = 0.929832
CACHED: /test2/fileNum=9 = 0.930613
CACHED: /test1/fileNum=21 = 0.93075
CACHED: /test1/fileNum=22 = 0.932898
CACHED: /test2/fileNum=91 = 0.934841
CACHED: /test1/fileNum=11 = 0.935731
CACHED: /test1/fileNum=12 = 0.936516
CACHED: /test1/fileNum=80 = 0.936531
CACHED: /test1/fileNum=35 = 0.93678
CACHED: /test1/fileNum=16 = 0.93743
CACHED: /test1/fileNum=7 = 0.938876
CACHED: /test3/fileNum=18 = 0.939387
CACHED: /test3/fileNum=76 = 0.939395
CACHED: /test2/fileNum=35 = 0.939418
CACHED: /test2/fileNum=23 = 0.939725
CACHED: /test2/fileNum=4 = 1
PURGE: Store size is 410/370,
        erase /test1/fileNum=1 value of 0.242089
PURGE: Store size is 370/370,
        erase /test3/fileNum=5 value of 0.467078
```

6.2.3 FATE QOS effect upon caching

listing 6.3 is the XML fragment for QOS matching. It looks for 3 QoS values in the packet and gives each a specific weight (QoS1 is 1) (QoS2 is .8) (QoS3 is .6). All other content will have a value of .4 (in reality, the formula is .38 + .02*rnd). Only packet /test1/fileNum=1 matches QoS1, /test2/fileNum=2 matches QoS2, and /test3/fileNum=3 matches QoS3. The FATE formulation becomes $MAX(QoS1 * .98, Qos2 * .78, QoS3 * .58, 0.38) + RND * 0.2$. Thus, any non QoS marked content will have a weight of 0.38, while other QoS values will be appropriately evaluated.

```
<Utility name="UTILITYBLOCK" order="1">
  <!-- COUNT QoS (count_condition=none)(matching_lower_bound=A
      matching_upper_bound=B); -->
  <Utility name="SUM">
    <Utility name="MAX">
      <Utility name="MULT">
        <Utility name="COUNT" count_condition="none" attribName="QOS"
            matching_lower_bound="1" matching_upper_bound="1"
            match_criteria="LeftRightInclusive"/>
        <Utility name="CONST" defaultValue="0.98"/>
      </Utility>
      <Utility name="MULT">
        <Utility name="COUNT" count_condition="none" attribName="QOS"
            matching_lower_bound="2" matching_upper_bound="2"
            match_criteria="LeftRightInclusive"/>
        <Utility name="CONST" defaultValue="0.78"/>
      </Utility>
      <Utility name="MULT">
        <Utility name="COUNT" count_condition="none" attribName="QOS"
            matching_lower_bound="3" matching_upper_bound="3"
            match_criteria="LeftRightInclusive"/>
        <Utility name="CONST" defaultValue="0.58"/>
      </Utility>
      <Utility name="CONST" defaultValue="0.38"/>
    </Utility>
    <Utility name="MULT">
      <Utility name="CONST" defaultValue="0.02"/>
      <Utility name="RND" randomType="alwaysRnd"/>
    </Utility>
  </Utility>
 </Utility>
```

Listing 6.3: QoS Packet XML format

QOS XML configuration snippet

Cache Snippet of QOS with alpha=0.8

QOS a0.8

```
CACHED: /test1/fileNum=5 = 0.381665
CACHED: /test1/fileNum=3 = 0.382289
CACHED: /test1/fileNum=79 = 0.382351
CACHED: /test2/fileNum=1 = 0.383386
CACHED: /test3/fileNum=1 = 0.384688
CACHED: /test3/fileNum=34 = 0.385511
CACHED: /test3/fileNum=12 = 0.38626
CACHED: /test3/fileNum=6 = 0.387108
CACHED: /test3/fileNum=16 = 0.393201
CACHED: /test1/fileNum=32 = 0.395977
CACHED: /test1/fileNum=10 = 0.398969
CACHED: /test1/fileNum=50 = 0.39917
CACHED: /test3/fileNum=67 = 0.39983
CACHED: /test3/fileNum=3 = 0.580155
CACHED: /test2/fileNum=2 = 0.782019
CACHED: /test1/fileNum=1 = 0.981504
PURGE: Store size is 400/370,
        erase /test1/fileNum=5 value of 0.381665
PURGE: Store size is 380/370,
        erase /test1/fileNum=3 value of 0.382289
```

Cache Snippet of QOS with alpha=1.2

QOS a1.2

```
CACHED: /test1/fileNum=12 = 0.382163
CACHED: /test2/fileNum=53 = 0.383754
CACHED: /test2/fileNum=60 = 0.383823
CACHED: /test3/fileNum=76 = 0.388166
CACHED: /test2/fileNum=4 = 0.389808
CACHED: /test2/fileNum=15 = 0.389939
CACHED: /test2/fileNum=3 = 0.390614
CACHED: /test3/fileNum=44 = 0.393168
CACHED: /test3/fileNum=18 = 0.394925
CACHED: /test3/fileNum=6 = 0.396114
CACHED: /test3/fileNum=5 = 0.39694
CACHED: /test2/fileNum=7 = 0.398771
CACHED: /test3/fileNum=2 = 0.399071
CACHED: /test3/fileNum=3 = 0.594125
CACHED: /test2/fileNum=2 = 0.784917
CACHED: /test1/fileNum=1 = 0.983106
PURGE: Store size is 400/370,
        erase /test1/fileNum=12 value of 0.382163
PURGE: Store size is 390/370,
        erase /test2/fileNum=53 value of 0.383754
PURGE: Store size is 380/370,
        erase /test2/fileNum=60 value of 0.383823
PURGE: Store size is 370/370,
        erase /test3/fileNum=76 value of 0.388166
```

6.2.4 FATE REGEX effects upon caching

Next is the XML snippet for a REGEX match, using the configuration in listing 6.4. If the field 'help' contains any match of 'SOS*', it will return a value of '1'. Only content /test1/fileNum=6, /test2/fileNum=5, and /test3/FileNum=4 contain this field. The FATE formula evaluation is $MAX(REGEX, RND * 0.2, PLE)$. Only new content and 'SOS' matching content will return a '1.0'.

```
<Utility name="UTILITYBLOCK" order="1">
    <Utility name="MAX">
     <Utility name="PLE"/>
     <Utility name="MULT">
       <Utility name="CONST" defaultValue="0.2"/>
       <Utility name="RND" randomType="alwaysRnd"/>
     </Utility>
       <!-- REGEX_MATCH -->
       <Utility name="REGEX_MATCH" matchFieldName="help"
           regexPattern="(SOS)(.*)">
     </Utility>
   </Utility>
 </Utility>
```

Listing 6.4: MAX(Regex,PLE) Packet XML format

REGEX XML configuration snippet

Cache Snippet of REGEX with alpha=0.8

REGEX_a0.8

```
CACHED: /test3/fileNum=12 = 0.0165522
CACHED: /test1/fileNum=1 = 0.0334573
CACHED: /test2/fileNum=1 = 0.0619461
CACHED: /test1/fileNum=2 = 0.081915
CACHED: /test3/fileNum=16 = 0.15571
CACHED: /test3/fileNum=67 = 0.179923
CACHED: /test3/fileNum=78 = 0.189962
CACHED: /test1/fileNum=50 = 0.192296
CACHED: /test3/fileNum=6 = 0.198144
CACHED: /test1/fileNum=10 = 1
CACHED: /test1/fileNum=6 = 1
CACHED: /test2/fileNum=5 = 1
CACHED: /test3/fileNum=4 = 1
PURGE: Store size is 380/370,
        erase /test3/fileNum=12 value of 0.0165522
PURGE: Store size is 370/370,
        erase /test1/fileNum=1 value of 0.0334573
```

Cache Snippet of REGEX with alpha=1.2

REGEX_a1.2

```
CACHED: /test1/fileNum=1 = 0.00616449
CACHED: /test2/fileNum=7 = 0.0167715
CACHED: /test2/fileNum=24 = 0.0234033
CACHED: /test2/fileNum=53 = 0.0715269
CACHED: /test1/fileNum=2 = 0.119695
CACHED: /test3/fileNum=6 = 0.165288
CACHED: /test3/fileNum=5 = 0.18933
CACHED: /test2/fileNum=3 = 0.192861
CACHED: /test1/fileNum=6 = 1
CACHED: /test2/fileNum=4 = 1
CACHED: /test2/fileNum=5 = 1
CACHED: /test3/fileNum=4 = 1
PURGE: Store size is 420/370,
        erase /test1/fileNum=1 value of 0.00616449
PURGE: Store size is 380/370,
        erase /test2/fileNum=7 value of 0.0167715
PURGE: Store size is 370/370,
        erase /test2/fileNum=24 value of 0.0234033
```

```
<Utility name="UTILITYBLOCK" order="1">
  <Utility name="MAX">
    <Utility name="SUM">
      <Utility name="MULT">
        <Utility name="RawEval" attribName="Distance"
            updateEntryPktType="8">
          <Normalize normalizeName="GeometricMatch" biasLowVal="false"
              invertValue="true" divisor="1"/>
        </Utility>
        <Utility name="CONST" defaultValue="0.9998"/>
      </Utility>
      <Utility name="MULT">
        <Utility name="CONST" defaultValue="0.0002"/>
        <Utility name="RND" randomType="alwaysRnd"/>
      </Utility>
    </Utility>
    <Utility name="PLE"/>
  </Utility>
</Utility>
```

Listing 6.5: MAX(Distance,PLE) Packet XML format

6.2.5 FATE Distance effects upon caching

This snippet deals with distance of producers from cache, using the configuration shown in listing 6.5. It is more costly, and requires more time, to fetch content from farther producers, than closer producers. The farthest distance is evaluated at higher, based upon the chosen normalizer, $1 - 1/n$. The producers have distance of 3, 7, and 10 hops away from the cache. The actual evaluation becomes $MAX((1 - DIST) * .9998 + 0.0002, PLE)$, so only new content is valued at '1'.

Distance XML configuration snippet

Cache Snippet of DISTANCE with alpha=0.8

DISTANCE a0.8

```
ACHED: /test1/fileNum=3 = 0.66664
CACHED: /test1/fileNum=13 = 0.666667
CACHED: /test3/fileNum=11 = 0.857005
CACHED: /test3/fileNum=29 = 0.857017
CACHED: /test3/fileNum=27 = 0.857037
CACHED: /test3/fileNum=4 = 0.857042
CACHED: /test3/fileNum=14 = 0.857059
CACHED: /test3/fileNum=8 = 0.857089
CACHED: /test3/fileNum=86 = 0.857103
CACHED: /test3/fileNum=60 = 0.857109
CACHED: /test3/fileNum=25 = 0.857137
CACHED: /test3/fileNum=2 = 0.857147
CACHED: /test3/fileNum=36 = 0.85715
CACHED: /test3/fileNum=7 = 0.857163
CACHED: /test2/fileNum=1 = 0.908917
CACHED: /test2/fileNum=62 = 0.90898
CACHED: /test2/fileNum=100 = 0.908997
CACHED: /test2/fileNum=35 = 0.909037
CACHED: /test2/fileNum=9 = 0.909047
CACHED: /test2/fileNum=2 = 0.909053
CACHED: /test2/fileNum=28 = 0.909057
CACHED: /test2/fileNum=14 = 0.909095
CACHED: /test2/fileNum=13 = 0.909095
CACHED: /test2/fileNum=25 = 1
PURGE: Store size is 380/370,
        erase /test1/fileNum=3 value of 0.66664
```

Cache Snippet of DISTANCE with alpha=1.2

DISTANCE a1.2

```
CACHED: /test1/fileNum=1 = 0.666725
CACHED: /test3/fileNum=76 = 0.857017
CACHED: /test2/fileNum=15 = 0.908915
CACHED: /test2/fileNum=1 = 0.908917
CACHED: /test2/fileNum=53 = 0.908925
CACHED: /test2/fileNum=5 = 0.908934
CACHED: /test2/fileNum=23 = 0.908947
CACHED: /test2/fileNum=9 = 0.908947
CACHED: /test2/fileNum=97 = 0.908994
CACHED: /test2/fileNum=12 = 0.908999
CACHED: /test2/fileNum=28 = 0.909008
CACHED: /test2/fileNum=4 = 0.909025
CACHED: /test2/fileNum=91 = 0.909053
CACHED: /test2/fileNum=10 = 0.909057
CACHED: /test2/fileNum=3 = 0.909059
CACHED: /test2/fileNum=24 = 0.909064
CACHED: /test2/fileNum=7 = 0.909069
CACHED: /test2/fileNum=60 = 0.909081
CACHED: /test2/fileNum=18 = 0.9091
CACHED: /test2/fileNum=84 = 0.909101
CACHED: /test2/fileNum=2 = 0.909107
CACHED: /test3/fileNum=44 = 1
PURGE: Store size is 380/370,
        erase /test1/fileNum=1 value of 0.666725
```

```
<Utility name="UTILITYBLOCK" order="1">
  <Utility name="MAX">
    <Utility name="SUM">
      <Utility name="MULT">
        <Utility name="RawEval" attribName="TotalSize">
          <Normalize normalizeName="GeometricMatch" biasLowVal="false"
              invertValue="false" divisor="10"/>
        </Utility>
        <Utility name="LRU">
          <Normalize normalizeName="NormalRanked" value_type="ceiling"/>
        </Utility>
        <Utility name="CONST" defaultValue="0.98"/>
      </Utility>
      <Utility name="MULT">
        <Utility name="CONST" defaultValue="0.02"/>
        <Utility name="RND" randomType="alwaysRnd"/>
      </Utility>
    </Utility>
    <Utility name="PLE"/>
  </Utility>
</Utility>
```

Listing 6.6: MAX(Size*Lru, PLE) Packet XML format

6.2.6 FATE SIZE*LRU effects upon caching

FATE will consider caching of content size with most requested content, using the configuration in listing 6.6. There are a myriad of ways to enact this type of cache, but for simplicity, it is simply multiplied together to determine it's valuation.

FATE SIZE*LRU configuration snippet

Cache Snippet of SIZE*LRU with alpha=0.8

SIZE*LRU a0.8

```
ACHED: /test1/fileNum=19 = 0.00150003
CACHED: /test3/fileNum=49 = 0.0648476
CACHED: /test1/fileNum=4 = 0.08568
CACHED: /test1/fileNum=46 = 0.102177
CACHED: /test3/fileNum=6 = 0.128193
CACHED: /test3/fileNum=78 = 0.139589
CACHED: /test1/fileNum=5 = 0.1401
CACHED: /test2/fileNum=4 = 0.159516
CACHED: /test1/fileNum=1 = 0.17647
CACHED: /test2/fileNum=2 = 0.183922
CACHED: /test1/fileNum=32 = 0.197466
CACHED: /test3/fileNum=34 = 0.238938
CACHED: /test3/fileNum=12 = 0.542932
CACHED: /test3/fileNum=67 = 0.632744
CACHED: /test3/fileNum=16 = 0.729498
CACHED: /test1/fileNum=50 = 0.898955
CACHED: /test1/fileNum=10 = 0.99292
CACHED: /test2/fileNum=1 = 1
PURGE: Store size is 410/370,
        erase /test1/fileNum=19 value of 0.00150003
PURGE: Store size is 400/370,
        erase /test3/fileNum=49 value of 0.0648476
PURGE: Store size is 390/370,
        erase /test1/fileNum=4 value of 0.08568
```

Cache Snippet of SIZE*LRU with alpha=1.2

SIZE*LRU a1.2

```
CACHED: /test2/fileNum=24 = 0.016996
CACHED: /test2/fileNum=1 = 0.0437696
CACHED: /test3/fileNum=6 = 0.0838327
CACHED: /test1/fileNum=4 = 0.123782
CACHED: /test2/fileNum=2 = 0.143654
CACHED: /test2/fileNum=3 = 0.150181
CACHED: /test1/fileNum=1 = 0.242025
CACHED: /test3/fileNum=76 = 0.330524
CACHED: /test3/fileNum=44 = 0.453642
CACHED: /test3/fileNum=5 = 0.504847
CACHED: /test1/fileNum=14 = 0.506578
CACHED: /test2/fileNum=7 = 0.881485
CACHED: /test2/fileNum=4 = 1
PURGE: Store size is 390/370,
        erase /test2/fileNum=24 value of 0.016996
PURGE: Store size is 380/370,
        erase /test2/fileNum=1 value of 0.0437696
```

6.2.7 FATE MAX(SIZE*LRU,QoS) effects upon caching

In this section, we present the configuration and cache contents of the $SIZE*LRU$ with respect to packets with correct QoS valuations, using the configuration in listing 6.7.

FATE MAX(SIZE*LRU,QoS) configuration snippet

Cache Snippet of MIN(SIZE*LRU,QOS) with alpha=0.8

*MIN(SIZE*LRU,QOS) a0.8*

```
CACHED: /test1/fileNum=73 = 0.0171146
CACHED: /test3/fileNum=28 = 0.0358469
CACHED: /test2/fileNum=8 = 0.0359898
CACHED: /test3/fileNum=17 = 0.107255
CACHED: /test2/fileNum=7 = 0.107398
CACHED: /test3/fileNum=6 = 0.122434
CACHED: /test2/fileNum=4 = 0.157155
CACHED: /test3/fileNum=21 = 0.178663
CACHED: /test1/fileNum=19 = 0.214082
CACHED: /test3/fileNum=49 = 0.250071
CACHED: /test1/fileNum=46 = 0.285633
CACHED: /test3/fileNum=78 = 0.32148
CACHED: /test1/fileNum=32 = 0.357041
CACHED: /test3/fileNum=34 = 0.392888
CACHED: /test3/fileNum=12 = 0.642816
CACHED: /test3/fileNum=67 = 0.714224
CACHED: /test3/fileNum=16 = 0.785633
CACHED: /test2/fileNum=2 = 0.797659
CACHED: /test1/fileNum=50 = 0.928306
CACHED: /test1/fileNum=1 = 0.992645
CACHED: /test1/fileNum=10 = 0.999714
CACHED: /test2/fileNum=1 = 1
PURGE: Store size is 400/370,
        erase /test1/fileNum=73 value of 0.0171146
PURGE: Store size is 390/370,
        erase /test3/fileNum=28 value of 0.0358469
PURGE: Store size is 380/370,
        erase /test2/fileNum=8 value of 0.0359898
PURGE: Store size is 370/370,
        erase /test3/fileNum=17 value of 0.107255
```

Cache Snippet of MIN(SIZE*LRU,QOS) with alpha=1.2

*MIN(SIZE*LRU,QOS) a1.2*

```
CACHED: /test1/fileNum=35 = 0.0908595
CACHED: /test3/fileNum=6 = 0.0909114
CACHED: /test2/fileNum=53 = 0.0914048
CACHED: /test1/fileNum=4 = 0.136326
CACHED: /test2/fileNum=3 = 0.170498
CACHED: /test2/fileNum=24 = 0.182264
CACHED: /test3/fileNum=76 = 0.454661
CACHED: /test3/fileNum=5 = 0.499909
CACHED: /test3/fileNum=44 = 0.545521
CACHED: /test1/fileNum=14 = 0.590769
CACHED: /test3/fileNum=3 = 0.599476
CACHED: /test2/fileNum=2 = 0.792411
CACHED: /test2/fileNum=7 = 0.90914
CACHED: /test1/fileNum=1 = 0.980482
CACHED: /test2/fileNum=4 = 1
PURGE: Store size is 410/370,
        erase /test1/fileNum=35 value of 0.0908595
PURGE: Store size is 400/370,
        erase /test3/fileNum=6 value of 0.0909114
```

```
<Utility name="UTILITYBLOCK" order="1">
  <Utility name="MAX">
    <Utility name="MULT">
      <Utility name="RawEval" attribName="TotalSize">
        <Normalize normalizeName="GeometricMatch" biasLowVal="false"
            invertValue="false" divisor="10"/>
      </Utility>
      <Utility name="LRU">
        <Normalize normalizeName="NormalRanked" value_type="ceiling"/>
      </Utility>
    </Utility>
    <Utility name="SUM">
      <Utility name="MAX">
        <Utility name="MULT">
          <Utility name="COUNT" count_condition="none"
              attribName="QOS" matching_lower_bound="1"
              matching_upper_bound="1"
              match_criteria="LeftRightInclusive"/>
          <Utility name="CONST" defaultValue="0.98"/>
        </Utility>
        <Utility name="MULT">
          <Utility name="COUNT" count_condition="none"
              attribName="QOS" matching_lower_bound="2"
              matching_upper_bound="2"
              match_criteria="LeftRightInclusive"/>
          <Utility name="CONST" defaultValue="0.78"/>
        </Utility>
        <Utility name="MULT">
          <Utility name="COUNT" count_condition="none"
              attribName="QOS" matching_lower_bound="3"
              matching_upper_bound="3"
              match_criteria="LeftRightInclusive"/>
          <Utility name="CONST" defaultValue="0.58"/>
        </Utility>
      </Utility>
      <Utility name="MULT">
        <Utility name="CONST" defaultValue="0.02"/>
        <Utility name="RND" randomType="alwaysRnd"/>
      </Utility>
    </Utility>
    <Utility name="PLE"/>
  </Utility>
</Utility>
```

Listing 6.7: MAX(Size*Lru, Qos1, Qos2, Qos3, PLE) Packet XML format

6.2.8 FATE MAX(SIZE*LRU,QOS,REGEX) effects upon caching

Extending the evaluation from only QoS customers, but high priority content, present the following FATE evaluation. The FATE cache valuation becomes $MAX(SIZE * LRU, MAX(QoS1 * .88, QoS2 * .78, QoS * .83) + RND * 0.2, REGEX * .98 + RND * 0.2)$ using the configuration defined in listing 6.8.

Cache Snippet of MIN(SIZE*LRU,QOS,REGEX) with alpha=0.8

*MIN(SIZE*LRU,QOS,REGEX) a0.8*

```
CACHED: /test3/fileNum=6 = 0.139026
CACHED: /test2/fileNum=7 = 0.832273
CACHED: /test3/fileNum=21 = 0.845669
CACHED: /test1/fileNum=19 = 0.852326
CACHED: /test3/fileNum=49 = 0.859091
CACHED: /test1/fileNum=46 = 0.865775
CACHED: /test3/fileNum=78 = 0.872513
CACHED: /test1/fileNum=32 = 0.879198
CACHED: /test1/fileNum=1 = 0.88216
CACHED: /test3/fileNum=34 = 0.885936
CACHED: /test3/fileNum=12 = 0.932915
CACHED: /test3/fileNum=67 = 0.946337
CACHED: /test3/fileNum=16 = 0.959759
CACHED: /test1/fileNum=6 = 0.98102
CACHED: /test1/fileNum=50 = 0.986578
CACHED: /test2/fileNum=5 = 0.989384
CACHED: /test3/fileNum=4 = 0.999372
CACHED: /test1/fileNum=10 = 1
PURGE: Store size is 380/370,
        erase /test3/fileNum=6 value of 0.139026
```

Cache Snippet of MIN(SIZE*LRU,QOS,REGEX) with alpha=1.2

*MIN(SIZE*LRU,QOS,REGEX) a1.2*

```
CACHED: /test2/fileNum=3 = 0.228919
CACHED: /test3/fileNum=5 = 0.499976
CACHED: /test3/fileNum=18 = 0.710837
CACHED: /test1/fileNum=35 = 0.758975
CACHED: /test2/fileNum=53 = 0.759071
CACHED: /test2/fileNum=24 = 0.783164
CACHED: /test3/fileNum=3 = 0.840952
CACHED: /test3/fileNum=76 = 0.855394
CACHED: /test3/fileNum=44 = 0.879487
CACHED: /test1/fileNum=14 = 0.891486
CACHED: /test1/fileNum=1 = 0.895248
CACHED: /test2/fileNum=7 = 0.975907
CACHED: /test2/fileNum=5 = 0.981427
CACHED: /test1/fileNum=6 = 0.983123
CACHED: /test3/fileNum=4 = 0.996156
CACHED: /test2/fileNum=4 = 1
PURGE: Store size is 410/370,
        erase /test2/fileNum=3 value of 0.228919
PURGE: Store size is 370/370,
        erase /test3/fileNum=5 value of 0.499976
```

```
<Utility name="UTILITYBLOCK" order="1">
  <Utility name="MAX">
    <Utility name="MULT">
      <Utility name="RawEval" attribName="TotalSize">
        <Normalize normalizeName="GeometricMatch" biasLowVal="false"
            invertValue="false" divisor="10"/>
      </Utility>
      <Utility name="LRU">
        <Normalize normalizeName="NormalRanked" value_type="ceiling"/>
      </Utility>
    </Utility>
    <Utility name="SUM">
      <Utility name="MAX">
        <Utility name="MULT">
          <Utility name="COUNT" count_condition="none"
              attribName="QOS" matching_lower_bound="1"
              matching_upper_bound="1"
              match_criteria="LeftRightInclusive"/>
          <Utility name="CONST" defaultValue="0.88"/>
        </Utility>
        <Utility name="MULT">
          <Utility name="COUNT" count_condition="none"
              attribName="QOS" matching_lower_bound="2"
              matching_upper_bound="2"
              match_criteria="LeftRightInclusive"/>
          <Utility name="CONST" defaultValue="0.78"/>
        </Utility>
        <Utility name="MULT">
          <Utility name="COUNT" count_condition="none"
              attribName="QOS" matching_lower_bound="3"
              matching_upper_bound="3"
              match_criteria="LeftRightInclusive"/>
          <Utility name="CONST" defaultValue="0.83"/>
        </Utility>
        <Utility name="MULT">
          <!-- REGEX_MATCH -->
            <Utility name="REGEX_MATCH" matchFieldName="help"
                regexPattern="(SOS)(.*)"/>
          <Utility name="CONST" defaultValue="0.98"/>
        </Utility>
      </Utility>
      <Utility name="MULT">
        <Utility name="CONST" defaultValue="0.02"/>
        <Utility name="RND" randomType="alwaysRnd"/>
      </Utility>
    </Utility>
    <Utility name="PLE"/>
  </Utility>
</Utility>
```

Listing 6.8: max(LRU*size,qos1,qos2,qos, regex, PLE) Packet XML format

6.2.9 FATE MAX(SIZE*LRU*distance,QOS,REGEX) effects upon caching

The final FATE QoS caching formula is represented by: $MAX(SIZE * LRU * (1 - DISTANCE), MAX(QoS1 * .88, QoS2 * .78, QoS * .83) + RND * 0.2, REGEX * .98 + RND * 0.2)$ using the listing in 6.9.

Again, there are myriad variations of how best to represent the desired outcome, but FATE shows its flexibility, by simple modifications to its formulaic evaluation to quickly and easily represent the desired outcome.

max(LRU*size*distance, qos1,qos2,qos, regex, PLE)

Cache Snippet of MIN(SIZE*LRU*DISTANCE,QOS,REGEX) with alpha=0.8

*MIN(SIZE*LRU*DIST,QOS,REGEX) a0.8*

```
CACHED: /test1/fileNum=50 = 0.00926431
CACHED: /test3/fileNum=6 = 0.111221
CACHED: /test2/fileNum=7 = 0.739798
CACHED: /test3/fileNum=12 = 0.746332
CACHED: /test3/fileNum=67 = 0.75707
CACHED: /test3/fileNum=16 = 0.767808
CACHED: /test2/fileNum=2 = 0.798336
CACHED: /test3/fileNum=3 = 0.847713
CACHED: /test1/fileNum=1 = 0.882126
CACHED: /test2/fileNum=5 = 0.98754
CACHED: /test1/fileNum=6 = 0.995815
CACHED: /test3/fileNum=4 = 0.995937
CACHED: /test1/fileNum=10 = 1
PURGE: Store size is 380/370,
        erase /test1/fileNum=50 value of 0.00926431
PURGE: Store size is 370/370,
        erase /test3/fileNum=6 value of 0.111221
```

```
<Utility name="MAX">
  <Utility name="MULT">
    <Utility name="RawEval" attribName="TotalSize">
      <Normalize normalizeName="GeometricMatch" biasLowVal="false"
          invertValue="false" divisor="10"/>
    </Utility>
    <Utility name="LRU">
      <Normalize normalizeName="NormalRanked" value_type="ceiling"/>
    </Utility>
    <Utility name="RawEval" attribName="Distance"
        updateEntryPktType="8">
      <Normalize normalizeName="GeometricMatch" biasLowVal="true"
          invertValue="true" divisor="1"/>
    </Utility>
  </Utility>
  <Utility name="SUM">
    <Utility name="MAX">
      <Utility name="MULT">
        <Utility name="COUNT" count_condition="none"
            attribName="QOS" matching_lower_bound="1"
            matching_upper_bound="1"
            match_criteria="LeftRightInclusive"/>
        <Utility name="CONST" defaultValue="0.88"/>
      </Utility>
      <Utility name="MULT">
        <Utility name="COUNT" count_condition="none"
            attribName="QOS" matching_lower_bound="2"
            matching_upper_bound="2"
            match_criteria="LeftRightInclusive"/>
        <Utility name="CONST" defaultValue="0.78"/>
      </Utility>
      <Utility name="MULT">
        <Utility name="COUNT" count_condition="none"
            attribName="QOS" matching_lower_bound="3"
            matching_upper_bound="3"
            match_criteria="LeftRightInclusive"/>
        <Utility name="CONST" defaultValue="0.83"/>
      </Utility>
       <Utility name="MULT">
        <!-- REGEX_MATCH -->
          <Utility name="REGEX_MATCH" matchFieldName="help"
              regexPattern="(SOS)(.*)"/>
        <Utility name="CONST" defaultValue="0.98"/>
      </Utility>
    </Utility>
    <Utility name="MULT">
      <Utility name="CONST" defaultValue="0.02"/>
      <Utility name="RND" randomType="alwaysRnd"/>
    </Utility>
  </Utility>
  <Utility name="PLE"/>
 </Utility>
</Utility>
```

Listing 6.9: max(LRU*size*distance, qos1,qos2,qos, regex, PLE) Packet XML format

Cache Snippet of MIN(SIZE*LRU*DISTANCE,QOS,REGEX) with alpha=1.2

*MIN(SIZE*LRU*DIST,QOS,REGEX) a1.2*

```
CACHED: /test2/fileNum=3 = 0.203483
CACHED: /test3/fileNum=5 = 0.399981
CACHED: /test2/fileNum=53 = 0.67473
CACHED: /test3/fileNum=76 = 0.684316
CACHED: /test2/fileNum=24 = 0.696146
CACHED: /test3/fileNum=44 = 0.70359
CACHED: /test2/fileNum=2 = 0.780773
CACHED: /test3/fileNum=3 = 0.842415
CACHED: /test2/fileNum=7 = 0.867473
CACHED: /test1/fileNum=1 = 0.898962
CACHED: /test2/fileNum=5 = 0.982971
CACHED: /test1/fileNum=6 = 0.99202
CACHED: /test3/fileNum=4 = 0.992062
CACHED: /test2/fileNum=4 = 1
PURGE: Store size is 410/370,
        erase /test2/fileNum=3 value of 0.203483
PURGE: Store size is 370/370,
        erase /test3/fileNum=5 value of 0.399981
```

6.3 Conclusion

Complex QOS caching can be easily achieved by using weighted atomic evaluators, in an algebraic expression. FATE easily allows the changes to be made for custom QOS caching evaluation. As shown, the desired QOS effect can be achieved by simple changes to the evaluators, normalizers, and algebraic formula. The strength of FATE comes from allowing, normally incompatible algorithm evaluations to be mathematically defined, allowing a simple algebraic formula to represent complex evaluations and outcomes.

Chapter 7

Hashed Caching With Fate

7.1 Introduction

Traditional internet caching uses ICP (Internet Caching Protocol) [103] to distribute HTTP content. ICP will query other web caches for content upon a miss. If one of the other caches has the desired content, the cache will request a copy to itself. It requires a query to other CDN's (Content Delivery Networks) on a miss, and does not scale well. Either the scale of other CDN caches must be kept small, or the algorithm becomes bloated with too many requests to other CDN caches. Some, such as Fan [40], recommend a bloom filter to minimize requests, and use the bloom filter to represent files present in each CDN cache. Others have suggested different distributed cache protocols. Karger, et al [55] suggested using 'random cache trees', which are load balanced, and hashed and accessed via a random function. ICN caching, on the other hand, is dependent upon on-path caches for easily accessible content, without requiring additional control protocols such as ICP. But, according to Dabirmoghaddam [33], on-path caching offers little benefit compared to caching near the edge. Off-path caching, of the variety used by CDN's, is difficult. An ideal solution for ICN would be a distributed

off-path cache (preferably closer to the edge), but not having a control protocol. FATE makes this possible for both ICN and HTTP content. FATE allows an Ipv4 packet to be forwarded to intermediate destinations. ICN, traditionally, uses on-path caching, which can be inefficient, especially if not traversed. FATE allows a chained IP field, for matching intermediate destinations. A traditional consumer-producer pair arrives at a FATE node. The FATE node will lookup the destination, and find an appropriate intermediate off path cache nodes. FATE will push the producer destination on a chained list in the FATE packet, and push the intermediate destination as the packet new destination. As each packet arrives at an intermediate destination, the list is popped off.

7.2 Redirect to Off-Path Cache

Typically, packets are routed along a path: Original Packet Source(S) -> Destination (D)

FATE will transmit the packet normally, except it will add an intermediate off-path cache destination. The intermediate destination is calculated by hashing the packet, and using the hashed value to determine which off-path node for the intermediate destination. A Packet arrives at a FATE node, and based upon the destination, intermediate nodes (A) are assigned (off path caches). The packet pushes the Destination onto the packet list of destinations, and puts the first intermediate route on the destination field.

Updated packet Source(S)-> Destination (A) Intermediate Path Field: A,D

When arriving at Node A, the next intermediate path is switched to the destination:

Node A: Source (S) -> Destination (D) Intermediate Path Field: D

This continues until the packet either gets a cache hit (and returns early) or

arrives at the producer destination.

The various FATE packets are modified to track current and future intermediate nodes. The interest packet is modified as shown in listing 7.1, and the return packet is appropriately modified as shown in 7.2.

In addition, the visited Nodes are pushed on a return path. This ensures, if a packet goes to the producer, it will have the correct off-path return path of the caches, before returning to the consumer.

By using a hash of the content name, the packet is redirected from its server to an appropriate off path cache.

In traditional internet routing, NDS (name domain servers) play the role of directing the desired named content to CDN's or to the original origin/producer of the content. When CDN's are used, the information of the desired producer is located inside the HTML header, allowing the CDN to fetch unavailable content. This requires NDS to not have poisoned entries, and a 3rd party vendor to handle sensitive content.

```
PktToFate:<FATEPKT purpose="4" name="/test1/fileNum=1/segment=0">
  <Attribute name="DstChain" nameType="1" dataType="1" data="10.0.0.13;" />
  <Attribute name="ReturnChain" nameType="2" dataType="1" data="" />
  <TempAttribute name="Ipv4Dst" nameType="10" dataType="1" data="10.0.0.10"
      />
  <TempAttribute name="Ipv4Src" nameType="11" dataType="1" data="10.0.0.1"
      />
  ...
</FATEPKT>

<FATEPKT purpose="4" name="/test1/fileNum=1/segment=0">
  <Attribute name="DstChain" nameType="1" dataType="1"
      data="10.0.0.13;10.0.0.10;" />
  <Attribute name="NAMECHAIN" nameType="17" dataType="1" data="Node1 ;
      " />
  <Attribute name="ReturnChain" nameType="2" dataType="1" data="" />
  <TempAttribute name="Ipv4Dst" nameType="10" dataType="1" data="10.0.0.13"
      />
  <TempAttribute name="Ipv4Src" nameType="11" dataType="1" data="10.0.0.1"
      />
  ...
</FATEPKT>
```

Listing 7.1: FATE interest packet being modified to an intermediate destination

```
Return Data Packet:
<FATEPKT purpose="8" name="/test1/fileNum=1/segment=0">
  <Attribute name="DATA" nameType="19" dataType="1" data="XX..." />
  <Attribute name="DstChain" nameType="1" dataType="1"
      data="10.0.0.13;10.0.0.1;"/>
  <Attribute name="NAMECHAIN" nameType="17" dataType="1"
      data="Node1 ; Node4 ;Node4 ; Node5 ; Node3 ; " />
  <Attribute name="ReturnChain" nameType="2" dataType="1" data="" />
  <Attribute name="ServerHitNodeName" nameType="20" dataType="4"
      data="3" />
  <Attribute name="Timestamp" nameType="4" dataType="2"
      data="0094357700000000" />
  <Attribute name="TtlHop" nameType="3" dataType="4" data="124" />
  <TempAttribute name="Ipv4Dst" nameType="10" dataType="1"
      data="10.0.0.13" />
  ...
</FATEPKT>
```

Listing 7.2: FATE data packet being modified to return to source, same path

Simple example of a normal consumer to producer request, with FATE packet modifications, where there is no on-path cache in Figure 7.1. FATE allows alternative routes, including towards on-path cache nodes in Figure 7.2, and if it misses, use that route to the producer (and return path through the cache node) in Figure 7.3. Listings 7.1 and 7.2 show the FATE packet modifications used for internodal routing.

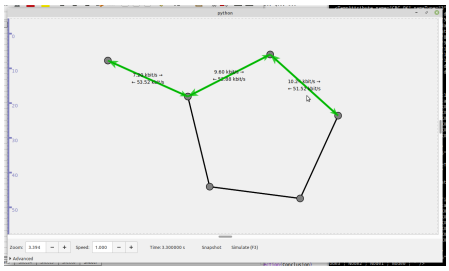

Figure 7.1: The original request path from consumer to producer

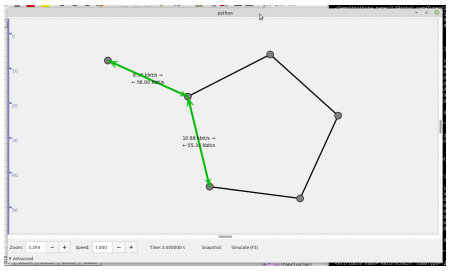

Figure 7.2: Simple redirected off-path cache hit

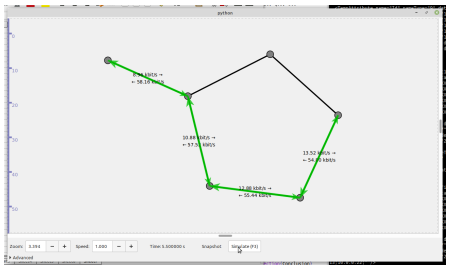

Figure 7.3: Simple redirected off-path server hit

7.3 Partition-Hashed Cache

In reality, cache locations may not be optimal, and may not be easily upgraded. Adding a new cache on a different network node, rarely helps increase overall cache-ability, typically requiring a replacement of the smaller sized cache with larger size cache. Using FATE, we look at a single, large contiguous cache (of 30 elements) and compare it to a distributed hash cache (3 nodes, with capacity of 10 elements per node). Typically, on-path caching is ineffective and suffers from a significant decay in cache hits the farther from the consumer. FATE attempts to resolve that problem by hashing the content, to provide a distinct partition for each content. Providing an 'extra' cache node is similar to hard drive striping, where each drive has a hashed portion of content. Whereas hard drives must be located in the same machine to be effective, that stipulation does not exist for network drives or caches.

For the single cache node, LRU is used to dictate eviction policy. In the hash-cached modules, the formula becomes $LRU * HASH(= num)$, where 'num' represents a matching modulus value (1,2,3) for the respective cache node.

7.4 Results

Simple example of single cache vs hashed cache (over 3 nodes):

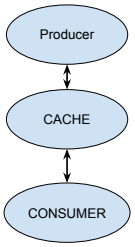

Figure 7.4: Single cache

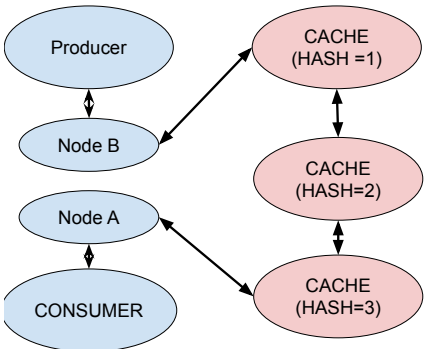

Figure 7.5: Simple 3-cache hashed network

Table 7.1: Total Cache vs split hash distributed caching

Alpha=1.2 Cache Topology	run1	run2	run3	run4	avg	std dev
Cache-30	20.03%	20.13%	19.72%	19.63%	19.83%	0.0022
3 Hash Cache 10	19.42%	19.52%	19.5%	19.25%	19.42%	0.0017
effectiveness	96.96%	96.97%	98.91%	98.03%	97.96%	0.0017

Simple Hashed Caching: 3x10 vs 30 results, N=100k LRU, 7200 sec, 20 req/sec, alpha=1

During the first run, the first cache had 1865 hits, second had 8798 hits, and the third had 3316 hits, out of 71970 requests.

Below shows an example of cache hits from hashing by name, and which cache contained it. Under that, we show the top ten cache distributions compared to the calculated distribution by cache type. It should be noted cache1 (from hash = 1) has the highest ratio of the top ten objects (66%), cache2 has 22% of the distribution, and cache0 has 12%. Hash by name happened to hash more to cache1 than cache2 or cache0. Based upon the calculations for a summation of the first 10 elements only, the expected weighted values closely match the actual percentage distributed to cache nodes at 63% for cache1, 24% for cache2, and 13% for cache0.

```
Top content, hashed value, and cache matching:
/test1/fileNum=1/segment=0 original hash:3636636289, to 1 [1,1]=1
/test1/fileNum=2 original hash:1685425049, to 2 [2,2]=1
/test1/fileNum=3 original hash:1398045355, to 1 [1,1]=1
/test1/fileNum=4 original hash:239951871, to 0 [0,0]=1
/test1/fileNum=5 original hash:84233572, to 1 [1,1]=1
/test1/fileNum=6 original hash:3591469750, to 1 [1,1]=1
/test1/fileNum=7 original hash:1621263263, to 2 [2,2]=1
/test1/fileNum=8 original hash:142833049, to 1 [1,1]=1
/test1/fileNum=9 original hash:4081996468, to 1 [1,1]=1
/test1/fileNum=10 original hash:2133279732, to 0 [0,0]=1
```

If we a rough, partial calculation for zipf distribution with only ten objects, we have:

Hash Results Top 10

```
summation of N=10, alpha=1.0 = 2.929

Position 1    0.341417152147406
Position 2    0.170708576073703
Position 3    0.113805717382469
Position 4    0.085354288036851
Position 5    0.068283430429481
Position 6    0.056902858691234
Position 7    0.048773878878201
Position 8    0.042677144018426
Position 9    0.03793523912749
Position 10   0.034141715214741
```

Using the hash function on the first ten objects, into three separate hashed receptacles shows it matches our hashed-cache distribution of the top ten items:

Hash Distribution

```
Hash 0    Hash 1    Hash 2
11.95\%    66.10\%    21.95\%    Top 10 Cache distribution by hash
13.34\%    62.94\%    23.72\%    expected hits by hash distribution
                                        of top 10 elements
```

Using a distributed cache methodology, a highly comparable cache hit-rate can be achieved as using a single higher capacity cache. In this test, a single 30 entry cache vs three 10 entry caches produced a 96+% effective cache rate. This allows easier expansion of caches using a distributed method, as it is easier to add additional caches compared to increasing the size of an existing cache. While on-path caching will not be as effective as a single large cache, as-is, it can become very competitive with proper tweaking of eviction criteria (as shown in this case, using a HASH to partially dictate caching).

7.5 Ubiquitous Hashed Caching

Using Intermediate Route Caching presented earlier, on a complex map with 160 nodes, 60 consumers and 10 producers (6 consumers will request the same content), and a variable number of caching nodes (10 and 83 caches).

Three scenarios are presented:

First, Intermediate Routing is disabled. This is the traditional IPv4/ICN routing, with a small possibility of encountering a randomly placed, on-path cache.

Second, Intermediate (Directed) Routing using content dedicated caches. When a FATE node assigns an intermediate cache, it is hashed from the producer destination address. In our scenario, with 10 caching nodes, every 6 consumers will request content from the same producer. Each FATE node will hash to the same intermediate off-path cache. These cache nodes are not exclusive to the content, they will cache any content which is off-path (via intermediate routing) or on-path (via random network routing through the node).

Lastly, Hashed-Directed Intermediate Routing works differently. The packet's ICN name is hashed and directed to a cache. Thus different packet names, going to the same producer can be directed to different caches. Each cache node reports how much free space is available. The entire free space is summed up,and a hash function modulus the availability is performed to determine which cache node destination is used.

7.6 Results

The results between on-path, hashed and content-centric are shown, with 10 and 83 cached nodes.

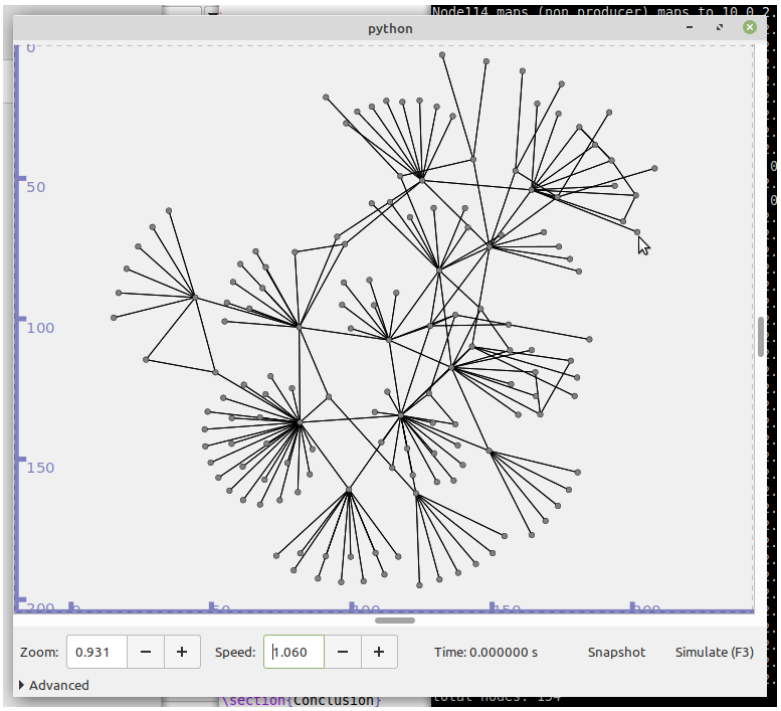

Figure 7.6: Complex redirected off-path server hit

Table 7.2: Offpath vs On-path: 10 caches, 10 producers, 60 consumers, N=100k

Alpha=1.2 Cache Topology	run1	run2	run3	run4	avg	std dev
Hash-Directed	28.84%	28.86%	28.82%	28.66%	28.8%	0.0009
Directed	28.84%	25.05%	23.89%	28.10%	26.47%	0.0238
On-Path	0%	2.94%	6.25%	0.51%	2.42%	0.0285
Alpha=0.8						
Hash-Directed	0.98%	0.97%	0.98%	0.99%	0.98%	0.0001
Directed	0.98%	0.84%	0.79%	0.95%	0.89%	0.0009
On-Path	0%	0.08%	0.18%	0.02%	0.07%	0.0008
Alpha=0.5						
Hash-Directed	0.031%	0.029%	0.034%	0.028%	0.031%	2.76e-05
Directed	0.031%	0.028%	0.028%	0.026%	0.028%	2.18e-05
On-Path	0%	0.002%	0.06%	0%	0.002%	2.71e-05

Table 7.3: Offpath vs On-path: Ubiquitous (83) caches, 10 producers, 60 consumers

Alpha=1.2 Cache Topology	run1	run2	run3	run4	avg	std dev
Hash-Directed	51.95%	42.78%	54.25%	36.56%	46.39%	0.0821
Directed	54.22%	51.05%	53.72%	52.85%	52.96%	0.0139
On-Path	0%	13.4%	12.36%	11.96%	9.43%	0.0631
Alpha=0.8						
Hash-Directed	5.59%	7.76%	5.81%	5.6%	6.19%	0.0105
Directed	4.98%	4.8%	5.12%	5.06%	4.99%	0.0014
On-Path	0.16%	0.39%	0.34%	0.33%	0.31%	0.0010
Alpha=0.5						
Hash-Directed	1.38%	4.86%	3.71%	6.12%	4.02%	0.0201
Directed	0.22%	0.21%	0.22%	0.2%	0.21%	0.0001
On-Path	0.01%	0.01%	0.01%	0.01%	0.01%	0

7.7 Comparison of Fate vs Traditional Caching

7.7.1 Traditional Caching

Caching, on the internet, has been extensively researched. Most of the recent work improved cache performance using a constraint of popularity. The problem is twofold: Popularity is not the only metric in a modern CDN. Customer level (QoS), size of content, time to first byte for a cache miss (farther content origins increase time) are all considered and measured. Yet, LRU is used not because it is the best popularity caching algorithm (it is not), but because of the ease of use, complexity of multiple constraints and temporal properties (which better algorithms, such as LFU, lack). At times, popularity is not the most important metric. CBMEN (DARPA program) used pocket networks of uniquely produced content. It was a problem of preserving unique content and distributed access to it.

Using traditional methods to (varying degrees) mimic FATE flexibility of

(MIN(SIZE * LRU * QoS, FRESHNESS):

1. Over all cached elements, evaluate all:
2. SIZE, LRU: MAP/SKIP LIST to find content quickly. Linked list to rank the content. Separate into multiple QoS buckets. (e.g., QoS1-Lru or QoS2-Lru)
3. FRESHNESS: Each content must be tagged with additional information
4. MULT: Use their relative position in the list to determine their rank or value.
5. MIN(): Compare if FRESHNESS is valid. If it is not, evict content. If it is valid, rank it against other content for eviction.
6. Problems: Inefficient traditional method: Requires O(N*E) ranking, where N is the number of content, and E are the number of evaluations.

A faster, but less flexible traditional method requires grouping everything into buckets:

1. TOTAL ORDERING SOLUTION: Have L*(E-1) number of buckets (L is the number of rankings, and E is the number of buckets). Rank each element to each other (e.g. QoS is lowest priority, then Size, and both are relatively ranked by LRU).
2. Identify the correct bucket based on the ordering (which subset of QoS, Size, ordered by LRU).
3. Take lowest ranked LRU in the appropriate Size-QoS buckets.
4. Evict from the lowest, non empty bucket.

5. Problems: Not Flexible, complex:

6. Requires time to create, program and debug each different structure for each caching constraint combination.

7. Adding new constraints makes the implementation much more difficult

8. Breaks down when you use different ordering algorithms (e.g. LRU and LFU).

9. Difficult to implement, when not fixed to a total ordering scheme for eviction.

7.7.2 FATE Caching

FATE, in caching, makes no assumptions. Fate resolves both issues of Caching. Each constraint, or desired ranking (such as popularity or freshness) is an independent evaluator. This evaluator returns, based upon its own method, a ranking of content: ranging from 0 to 1. As each value is a normalized scalar [0,1], the results can be weighted, multiplied, added, minimum, maximum, or other aggregate functions. This allows complex algorithmic implementations for requirements and allows quick implementation. If the desired evaluator does not exist, it must be created, which is much simpler in scope and complexity than a full caching algorithm. Each Evaluator, independently, evaluates each packet (QoS, LRU, SIZE, and Freshness). The results from each evaluator are weighted and combine into an algebraic expression. Extra flexibility and simplicity: If different weights or algebraic expression is desired, only the XML configuration file is changed (no compile) Existing formula is easily modified for new constraints (e.g. Distance from Producer) to adapt to different network systems

7.8 Conclusion

On-Path caching gets the lowest rate of cache hits. It shows the inefficiency of on-path routing. Hash-Directed caching performs better at lower alpha's (for zipf), while Directed caching performs the best for highly repetitive (high alpha) content.

Chapter 8

Functional Algebraic aTomic Evaluators in Packet Forwarding

8.1 Introduction

Developing a new network routing protocol can be an laborious process complicated by the constraints on how information is shared.

Each of these constraints affect the chosen routing algorithm: how is collision detected (if detected at all), and how is it handled? What is the best route, hop distance vs link bandwidth? Priority content vs low priority content, how is it handled? The answer, typically, is "It depends". Depending on what is desired, a longer optical fiber route may be preferred to over-the-air transmission. Battery life for mobile devices/sensors may limit what needs to be routed. FATE is similar to an FPGA; it is not as fast as a custom ASIC, but it has many advantages: it allows reprogramming in the event the internal algorithm is not optimal; it allows immediate testing of algorithms, as opposed to waiting for custom silicon to be delivered. When a potential design is hindered by multiple parameters, creating

multiple versions of the software, each version having different dependencies, will create unnecessary hardship in comparison of results, validation of algorithm, and later improvements. Creating new algorithms, and comparing them to existing research has many problems, including reproducibility [60] [97].

FATE evaluates information, from contextual metadata, to decide a course of action, based upon the evaluated results. In essence, instead of a dependency upon a fixed algorithmic implementation, FATE makes decisions based upon algebraic atomic evaluators. Each evaluator performs an atomic evaluation of information, which reflects the Linux philosophy of "Doing one thing, and do it well". The entire evaluation formula consists of the atomic evaluators, connected by algebraic expressions, to give a higher level evaluation (e.g. MIN(FN1, FN2, MAX((FN3*0.6+0.4*FN4), 0.4))). FATE, in networking, can evaluate various content to store/evict in a cache, determine which egress port offers the best performance (in terms of networking constraints), or which content to prefetch. The purpose of FATE is two fold. The first is to allow rapid changes to a given algorithm, due to algorithmic changes, erroneous assumptions, or different constraints. The second purpose is replicability of results regardless of architecture. FATE is agnostic to the types of networking environments (hardware or various simulators), or kernel/operating system implementations. FATE, as presented, allows faster development for custom routing protocols, and ensures reproducibility when compared on different platforms.

8.2 Related Work

8.2.1 NS3 Network Simulator

NS-3[31] is one of the most popular network simulators, with over a thousand papers published on its platform. NS3 is event driven, written in C++11 (same as FATE), to schedule networking events. NS3 also models real world interactions, such as error rates on various mediums, and collisions.

8.2.2 ENCODERS

ENCODERS[61, 95] (Edge Networking with Content-Oriented Declarative Enhanced Routing and Storage) is an SRI implementation of the PSIRP ICN models, based upon Haggle[87]. PSIRP and Encoders both use a bloom filter based Pub-Sub model, disseminating interests to neighbor nodes, and those neighbors return matches to the data. ENCODERS used a very early iteration of FATE called Utility Networking (Utility Caching-Prefetch, and Utility Forwarding), which evaluated content, and based upon that value, made a decision. In ENCODERS, Forwarding was fixed to several schemes, and was done via an early immature version of Utility Forwarding, Prophet[66], Direct[89], or Epidemic[99].

8.2.3 Traditional Routing

Traditional routing, such as IP (Internet Protocol), uses a routing table, consisting of subnets, which dictate the interface, gateway and next-hop neighbor for each forwarded packet. The tables are populated by various routing protocols to determine the best egress port, for packet delivery; typically shortest hop distance or largest path bandwidth. The tables are populated by various routing protocols, including but not limited to: OSPF (Open Shortest Path First) [74], RIP

(Routing Information Protocol [47], EIGRP (Enhanced Interior Gateway Routing Protocol) [15]. OLSR (Open Link State Routing) [29], and static routing. When a packet is ready to be transmitted, the Layer 2 medium is checked for in-progress packet transmissions. Typically, wired networks use CSMA/CD (packet collision) and wireless networks use CSMA/CA (use RTS/CTS or Ready/Clear signals to control when packets are sent) to avoid packet congestion/collisions in the network.

8.3 FATE Forwarding Implementation

FATE resolves system dependencies, such as timers or hardware (such as GPS), by using its own functions, wrapped and translated (if necessary) from the correct architectural implementation. Thus, a timer on the NS3 simulator will use the event-driven timer available in NS3, but in a Linux implementation, use the Linux timer POSIX methods. Mentioned for completeness, FATE uses a flexible packet framework, which uses a type-name-value tuple in the packet. FATE does use uniquely named information, or named data, to identify each unique chunk of data, allowing access to all the packet attributes by name.

FATE is organized in a top-down format, where the 'Node' module (8.3.2) handles specific purpose modules. Specific purpose modules can be 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, security, et al.

Each utility is an atomic, algebraic function, which 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 will take a valuation of the content, and perform an action on it (e.g. Caching will store or evict

content, forwarding will decide which packet for which egress port and next hop neighbor, or security will evaluate the packet for trust-worthiness). In this paper, we concentrate on the forwarding aspects of FATE.

FATE uses a modified BSD license. The license makes the code free to use, with the exception of giving credit when FATE is used in any manner.

8.3.1 Functional Algebraic aTomic Evaluators

The concept of FATE is to evaluate information (packet attributes, network conditions, or physical (PHY) properties), and perform an action, based upon said result. In order to evaluate a result, atomic algebraic functions are used. Each function can be an aggregate (such as minimum or addition), or an atomic evaluator.

Aggregation Functions

FATE supports several aggregation methods, all take one or more inputs, and returns an appropriate result. Below is a partial listing of available aggregation functions:

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

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

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

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

Atomic Functional Methods

FATE supports several atomic functional methods, below is a partial listing. Each atomic function may be stateful, but the state is exclusive to each instance

of the function. Atomic functions 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 to have many singular functions do the evaluation, then weighted based upon the appropriate aggregate function). Many functions are generic, with configuration options to allow them to be aliased to specific functionality. The following is a subset of atomic algebraic functions, currently available:

CONSTANT: A constant value, typically used with multiplication, e.g., 0.4 * TOS (Type Of Service).

PKT_ATTRIBUTE: This function relays a value of a specified packet attribute. Examples include TTL (which returns a '1' if TTL is 1+, but '0' if it is zero), TOS/COS (where a matching attribute value returns a predefined value, e.g., TOS of '3' returns '1.0', etc.). Some of the functions are generic, applied to many fields (e.g. TOS matches a specific value, it returns '1'. It depends on multiplying by a constant to give a weighted value, such as 0.4 * TOS. While the generic function uses a named field (all field attributes are accessed by name), they are, typically, written in a straightforward manner to identify their purpose. Thus, 'TOS' is shorthand for PKT_ATTRIBUTE("TOS", 3).

TABLE_ATTRIBUTE : ATTRIBUTE(node/PHY-name, name) is used when a function needs to access a table to determine a value. Examples of this are properties of the PHY (speed or properties, such as secure fiber optical cable vs wireless broadcasting), or a nodes' power measurement (e.g. how much battery is left). Other traditional attributes may be accessed from an internal node table, including if the medium is busy, PHY is transmitting, or measurements of network congestion (if available). Typically, discovery (or HELLO) packets are used to help

fill in tables, such as new neighbors, or monitor how often collisions occur.

TUPLE_NORMALIZED : TUPLE_NORMALIZE(PHY, Source, Destination, value, FN()) Table of tuples, which return a value. Typically, hop-counts for the specified PHY, and how many hops from 'Source' to 'Destination', are used as a key. Any tuple property can be modeled, but, for this paper, it is aliased to HOPCOUNT(). The actual normalization function is dictated by passing in a function, FN(). This allows a choice of linear, logarithmic, or other normalization of counts.

Packet Attributes: FATE packets can carry many attributes, most of which can be evaluated. There is an exception to this, currently being used to forward IP packets. By using a chain (list) of intermediate destinations, and allow the existing layer 3 (typically IP) to route the packet. FATE has its own packet format, and can be encapsulated within any L2/L3/L4 packet (or it can be left a pure FATE packet). But, this attribute can be used to encapsulate a list of Addresses (A,B,C,D). When a packet (such as IP) is sent and arrives at the correct destination, this field is checked, and the destination packet is changed. E.g. if a packet has a destination of E, when it arrives at E, it is replaced with 'A' (which is popped off the attribute queue). When the packet is received by 'A', it is forwarded to 'B' (as the new destination). There are multiple uses for this feature, but it does not evaluate as other atomic functions. This feature is used to query specific nodes, off-path caching, share information between specific network nodes, and avoid known down network pitfalls. The routing used is the original routing protocol, but simply forced to route multiple times, to collect or share information.

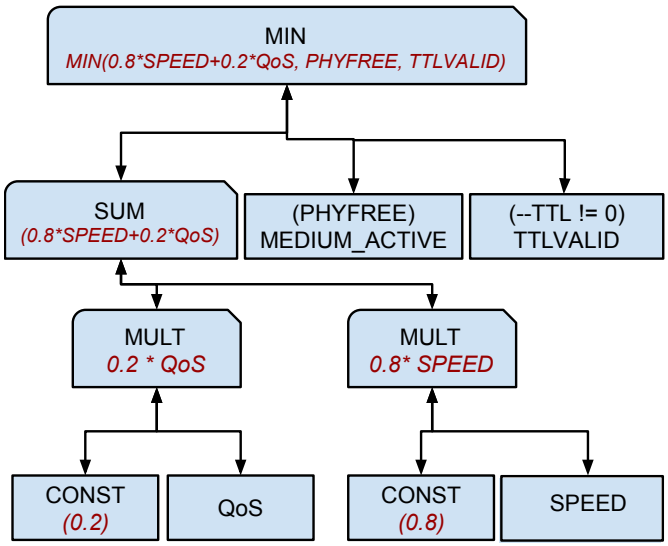

Figure 8.1: Simple forwarding representation of MIN(0.8*HOPCNT+0.2*QoS, PHYFREE, TTLVALID)

8.3.2 Modules

FATE is structured as a tree illustrated in Figure 8.1. Atomic functions are the leaves of the tree and algebraic aggregate functions are tree 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 three main (atomic) functions to forward, in this simple example: We take an evaluation of the HOPCNT (hop count to the destination, from a specific neighbor node and PHY), weighted at 80%, and the QoS (Quality of Service, typically TOS field in IP) at 20% weight (summing them). To prevent sending a packet when the TTL has expired, we do a MIN (minimum) with TTLVALID ('1' if the TTL field is 1+, otherwise '0'); note the Figure shows it being pre-decremented (some FATE utilities can modify the packet fields/attributes). In addition, we do not wish to transmit the packet when the PHY is busy, or the timer is still counting down from a packet collision, such as CSMA/CD or /CA (represented by PHYFREE). Based upon the evaluation, it is

up to the final evaluation to decide if the packet is forwarded, dropped, or must wait in the queue.

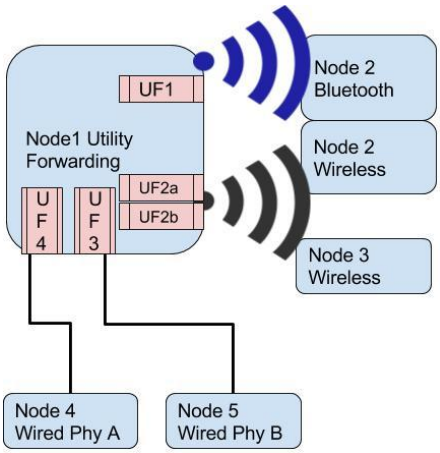

Figure 8.2: Utility forwarding tree representation of physical ports

When FATE evaluates a packet for forwarding, it evaluates against every PHY-Neighbor pair (against the destination node, not shown). As shown in Figure 8.2, UF1 has 1 neighbor node via bluetooth, UF2 has two WiFi neighbors, and UF3 and UF4 each have a single wired neighbor. Notice the UF1 and UF2 share the same neighbor, but using different PHYs. For each packet, the evaluation occurs on the UF1-Node2, UF2-Node2, UF2-Node3, UF2-Node4, UF2-Node5 pairs. Thus, in this case, for a single forwarding, 5 evaluations are made, and the highest rated evaluation wins. FATE allows a minimum threshold to send. In case there is more than a single packet to send, the first packet is sent out, and that PHY (and its neighbors) are removed from evaluation. Thus, if a queue of 2 packets are waiting to be routed, and the first is transmitted on UF2, the next packet is evaluated on UF1,UF3, and UF4. If any of those meet the minimum threshold to send (the threshold is set in the XML file), it will be sent on another PHY, to allow more efficient load balancing. As an example, WiFi might be preferred, but bluetooth could be acceptable for packet transmission. FATE configures at the

PHY level, sharing the same configuration for each PHY-Neighbor pair, but does not require all PHY configuration files to be the same (allowing flexibility in WiFi vs Ethernet collision avoidance/detection). When each neighbor is compared, the table lookup utility is used to identify (as a normalized vector) the distance to the intended destination.

8.4 Example Forward Load Balancing

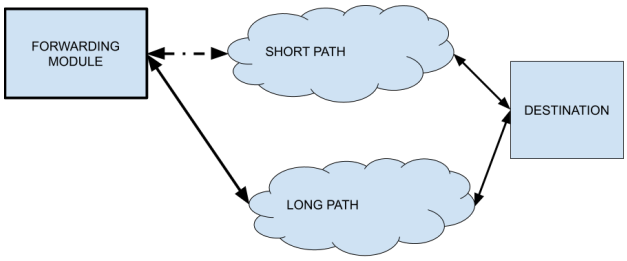

Figure 8.3: Example: FATE load balancing

Considering the following three constraints, and the desire is to minimize congestion by using the FATE formula: MIN(DIST * DELIVERTIME, TIMEOUT) Time to delivery: Delivery time from request to delivery. The prior packet's results are sent as a packet attribute, thus the Forwarding module gets the delivery time rate from the prior packets. Packet failure timeout (TIMEOUT): Using ACK's, measure the rate of successive and successful packet transmission. For the timeout, it returns '0', if the timeout expires, it returns '1'. This is per destination for ACKs, but apply universally if using Ethernet collision detection. Distance (Hops) to Destination (DIST) : Hops to destination DIST will use the distance normalizer (e.g. 10 hops = 0.5, 5 hops = 1.0), as data is more expensive when it must be retrieved over longer distances. DELIVERY TIME will use linear time normalization (similar to LRU) over 'x' seconds TIMEOUT will use exponential

timeout (2^n)microseconds (zero if no ACKs are seen), otherwise it will be '1'

Comparing two cases (SHORT1, LONG1 and SHORT2, LONG2), it can be seen how algebraic evaluators make a decision to forward.

Table 8.1: Sample PHY Table Setup for Node B

Constraint	SHORT1	LONG1	SHORT2	LONG2
Distance	1.0	0.5	1.0	0.5
Delivery Time	1.0	0.6	0.4	0.6
Timeout	0	1.0	1.0	1.0
MIN()	(1*1, .0)=.0	(.5*.6, 1)=.3	(1*.4, 1)=.4	(.5*.6,1)=.3

Another form of congestion is BUFFERBLOAT. When the packet buffers are full, it takes time for the buffers to drain. During this time, packets are dropped, which may affect one packet stream more than another. Simple BUFFERBLOAT solution: Drop the lowest ranked packet in buffer (similar to caching) Packet importance (IMPORTANCE) Customer origin (some customers pay to have higher QoS) (QOS) Packet Type (UDP is less sensitive to drops than TCP; network control packets may warrant higher importance) (TYPE) Quota per destination (shaping & sharing) (QUOTA): Ensure fair streaming bandwidth MIN((IMPORTANCE * QOS)*0.5 + TYPE*0.5), QUOTA)

Drop all packets which don't meet a certain threshold (e.g. 0.2), or lowest ranked packet, same as caching.

8.5 Sample Results

FATE is currently available, but the full integration with NS3 simulator is not yet complete (caching is complete, with partial routing). FATE is missing the NS3 integration to retrieve hop-counts (from IPv4), PHY speed, and collision rate from the NS3 simulator.

Figure 8.4 illustrates a sample network using Table 8.2 for the PHY properties of Node B and Table 8.3 for the neighbor $\leftrightarrow$ {destination hop count per phy-pair}.

As stated in Table 8.2, the value of the PHY network speed is logarithmic to the largest value (1Gbps) In a similar manner, the hop count is reduced to a normalized value between [0,1]. The value of a destination 1-hop away is '1.0', while three hops away is '.034'.

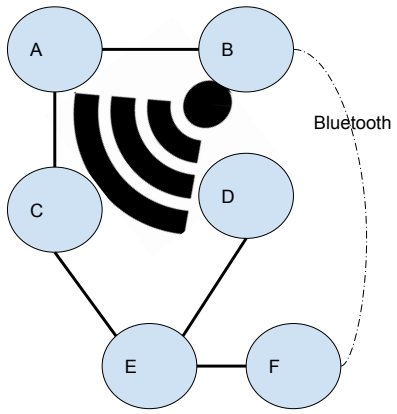

Figure 8.4: Representation of FATE with PHY-neighbor pairs

Using an algebraic formula of HOP*SPEED for route B to F, returns three values, the Bluetooth route (1 hop, lower speed; 0.23 * 1 = 0.23), or the wired route (3 hops, 1gbps speed; 0.34 * 1 = 0.34. From this formula, it is preferable to send packets over the longer but faster wired route. Another example is from node B to C, which gives a value, of 1 hop via WiFi of 1.0*0.67, as opposed to a wired route (2 hops) of 0.67*1. In this instance, both have an equal rating (ties can be broken by random determination, or by additional algebraic criteria).

Or, using TOS values to determine priority in a congested system, where TOS of 0 returns '0.2', and TOS (Type of Service) of 7 returns '1.0', we can a simple TOS*HOPS. This allows short path packets to be delivered despite their TOS

Table 8.2: Sample PHY Table Setup for Node B

PHY	Range	Speed	SpeedValue (logarithmic)
1-Wire	inf	1gbs	1
2-Blue-tooth	5m	5mbs	0.23
3-Wireless	60m	100mbs	0.67

Table 8.3: Sample Partial HOP Table Setup for Node B

PHY	Neighbor	Dest Node	Hops	HopValue
1	A	A	1	1
1	A	C	2	0.67
1	A	D	3	0.34
2	F	F	1	1
2	F	E	2	0.67
3	C	C	1	1
3	D	D	1	1
3	D	E	2	0.67

value, while longer packet paths are served only by higher value TOS. Assuming a maximum of 20 hops, a TOS(0), going 1 hop, will have the same value as a TOS(7) packet going 4 hops (TOS * HOPS; 0.2 * 1.0 for short path, 1.0 * 0.2(4/20)). There is no restriction to HOW the packet gets evaluated. To always give priority to TOS packet, routing can occur by TOS, then HOPS, e.g., TOS*0.8 + HOPS*0.2, or MIN(TOS, HOPS) * SPEED. Obviously, these values are chosen to give a sense of how a route would be selected, by evaluation, not an actual forwarding algorithm per se.

8.6 Conclusion

Reducing algorithmic functionality to specialized atomic functions greatly increases the flexibility and power of new implementations of routing protocols. As

each atomic evaluation function is created, it adds to the basic building blocks for succeeding algorithms, and enables designers to focus on rapid and efficient algorithmic development. The plug-and-play aspects of FATE significantly reduce development time, and turn complex algorithms to simple algebraic expressions (configured by XML). FATE forwarding is meant to be a consistent cross platform and allow rapid development. It still keeps internal state and processes each packet to every PHY-Neighbor (to destination) pair. Typically, FATE uses more CPU and is slower than the equivalent optimized algorithm. FATE is implemented under a modified BSD license, with the only requirement of giving credit for usage of the code or concepts therein. The code is available online [68].

Chapter 9

Conclusion

This thesis introduced FATE, a simple approach for more efficient and effective data interpretation in protocol stacks. It consists of a group of data interpreters, or experts, that output the utility of an input piece of data based on an algebraic equation defined for a protocol and those outputs can be combined into an aggregated utility that reflects a complex statement regarding the policies and restrictions with which a protocol uses the data at hand to take actions.

FATE is configurable to handle changes in the network topology or user requirements. It was shown that FATE can be used to improve the performance of caching systems and handle multiple quality-of-service constraints. It was also shown that FATE can be used for data forwarding subject to multiple constraints route over many constraints. FATE was also shown to support hash-based routing, where an Interest packet can hash the data name to resolve to a distributed cache network. In this manner, both routing and caching are used to reduce producer load and improve consumer response time for content delivery.

This thesis was limited to static functions and policies defining the way in which the utility of data is computed for a protocol. However, a promising area of future work consists of making the algebraic functions and the methods used

to aggregate them change dynamically in the context of reinforcement learning to make the data interpretation component of a protocol more self-reliant,without having to change the basic operation of the protocol itself. Simple examples of how this could be approached are the parameter adaptations that have been proposed in the recent past for TCP to take advantage of reinforcement learning in the computation of round-trip times.

Bibliography

[1] Content mediator architecture for content-aware networks - comet. Online; accessed 13-October-2015.

[2] Expressie internet architecture - xia. Online; accessed 13-October-2015.

[3] Green icn. Online; accessed 13-October-2015.

[4] Honeypot (computing). Online; accessed 13-October-2015.

[5] Personal and social communication services for health and lifestyle monitoring. Online; accessed 13-October-2015.

[6] Publish-subscribe internet technology - pursuit. Online; accessed 13-October-2015.

[7] Publish-subscribe internetrouting paradigm - psirp. Online; accessed 13-October-2015.

[8] Scalable and adaptive internet solutions - sail. Online; accessed 13-October-2015.

[9] Taguchi methods. Online; accessed 13-October-2015.

[10] B Adamson, C Bormann, M Handley, and J Macker. Negative-acknowledgment (nack)-oriented reliable multicast (norm) protocol. Technical report, 2004.

[11] Alexander Afanasyev, Ilya Moiseenko, and Lixia Zhang. ndnSIM: NDN simulator for NS-3. Technical Report NDN-0005, NDN, October 2012.

[12] Deepali D Ahir and Sagar B Shinde. Caching simulators for content centric networking. *International Journal of Science and Research (IJSR)*, 2014.

[13] Bengt Ahlgren, Matteo D'Ambrosio, Marco Marchisio, Ian Marsh, Christian Dannewitz, Börje Ohlman, Kostas Pentikousis, Ove Strandberg, René Rembarz, and Vinicio Vercellone. Design considerations for a network of information. In *Proceedings of the 2008 ACM CoNEXT Conference*, page 66. ACM, 2008.

[14] Jeff Ahrenholz, Claudiu Danilov, Thomas R Henderson, and Jae H Kim. Core: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1–7. IEEE, 2008.

[15] R Albrightson, JJ Garcia-Luna-Aceves, and Joanne Boyle. Eigrp–a fast routing protocol based on distance vectors. 1994.

[16] Andrea Araldo, Michele Mangili, Fabio Martignon, and Dario Rossi. Cost-aware caching: optimizing cache provisioning and object placement in ICN. In *IEEE Globecom 2014*, Proceedings of Global Communications Conference (GLOBECOM), 2014 IEEE, pages 1108 – 1113, Austin, United States, December 2014.

[17] Nils Aschenbruck, Raphael Ernst, Elmar Gerhards-Padilla, and Matthias Schwamborn. Bonnmotion: a mobility scenario generation and analysis tool. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, page 51. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

[18] César Bernardini, Thomas Silverston, and Olivier Festor. Socially-aware caching strategy for content centric networking. In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.

[19] 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.

[20] Gabriel M. Brito, Pedro Braconnot Velloso, and Igor M. Moraes. *Main ICN Architectures*, pages 23–42. John Wiley and Sons, Inc., 2013.

[21] Guohong Cao, Liangzhong Yin, and Chita R Das. Cooperative cache-based data access in ad hoc networks. *Computer*, 37(2):32–39, 2004.

[22] Wei Koong Chai, Diliang He, Ioannis Psaras, and George Pavlou. Cache "less for more" in information-centric networks. In *NETWORKING 2012*, pages 27–40. Springer, 2012.

[23] Narottam Chand, Ramesh C Joshi, and Manoj Misra. Cooperative caching in mobile ad hoc networks based on data utility. *Mobile Information Systems*, 3(1):19–37, 2007.

[24] Lei Chen and Wendi B Heinzelman. Qos-aware routing based on bandwidth estimation for mobile ad hoc networks. *IEEE Journal on selected areas in communications*, 23(3):561–572, 2005.

[25] Raffaele Chiocchetti, Dario Rossi, and Giuseppe Rossini. ccnsim: An highly scalable ccn simulator. In *ICC*, pages 2309–2314. IEEE, 2013.

[26] Raffaele Chiocchetti, Davide Rossi, and Giuseppe Rossini. ccnsim: An highly scalable ccn simulator. In *Communications (ICC), 2013 IEEE International Conference on*, pages 2309–2314. IEEE, 2013.

[27] Kideok Cho, Munyoung Lee, Kunwoo Park, Ted Taekyoung Kwon, Yanghee Choi, and Sangheon Pack. Wave: Popularity-based and collaborative in-network caching for content-oriented networks. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*, pages 316–321. IEEE, 2012.

[28] Hoon-gyu Choi, Jungmin Yoo, Taejoong Chung, Nakjung Choi, Taekyoung Kwon, and Yanghee Choi. Corc: coordinated routing and caching for named data networking. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 161–172. ACM, 2014.

[29] Thomas Clausen, Philippe Jacquet, Cédric Adjih, Anis Laouiti, Pascale Minet, Paul Muhlethaler, Amir Qayyum, and Laurent Viennot. Optimized link state routing protocol (olsr). 2003.

[30] Brian Cohen. Bit torrent. Online; accessed 13-October-2015.

[31] NS-3 Consortium. Network simulator ns3. Online; accessed 13-October-2015.

[32] 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.

[33] Ali Dabirmoghaddam, Maziar Mirzazad Barijough, and J.J. Garcia-Luna-Aceves. Understanding optimal caching and opportunistic caching at "the edge" of information-centric networks. In *Proceedings of the 1st ACM Conference on Information-Centric Networking*, ACM-ICN '14, page 47–56, New York, NY, USA, 2014. Association for Computing Machinery.

[34] Christian Dannewitz. Netinf: An information-centric design for the future internet. In *Proc. 3rd GI/ITG KuVS Workshop on The Future Internet*, 2009.

[35] Christian Dannewitz, Dirk Kutscher, Börje Ohlman, Stephen Farrell, Bengt Ahlgren, and Holger Karl. Network of information (netinf)–an information-centric networking architecture. *Computer Communications*, 36(7):721–735, 2013.

[36] Fahad R Dogar, Amar Phanishayee, Himabindu Pucha, Olatunji Ruwase, and David G Andersen. Ditto: a system for opportunistic caching in multi-hop wireless networks. In *Proceedings of the 14th ACM international conference on Mobile computing and networking*, pages 279–290. ACM, 2008.

[37] Stephanie Fraley et al. Taguchi methods using orthogonal arrays. Online; accessed 13-October-2015.

[38] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[39] Suyong Eum, Kiyohide Nakauchi, Masayuki Murata, Yozo Shoji, and Nozomu Nishinaga. Potential based routing as a secondary best-effort routing for information centric networking (icn). *Computer Networks*, 57(16):3154–3164, 2013.

[40] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293, 2000.

[41] Seyed Kaveh Fayazbakhsh, Yin Lin, Amin Tootoonchian, Ali Ghodsi, Teemu Koponen, Bruce Maggs, KC Ng, Vyas Sekar, and Scott Shenker. Less pain, most of the gain: Incrementally deployable icn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 147–158. ACM, 2013.

[42] Nikos Fotiou, Pekka Nikander, Dirk Trossen, and George C Polyzos. Developing information networking further: From psirp to pursuit. In *Broadband Communications, Networks, and Systems*, pages 1–13. Springer, 2012.

[43] JJ Garcia-Luna-Aceves, James Mathewson, Ram Ramanathan, and Bishal Thapa. Loop-free integrated forwarding and routing with gradients. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 1–9. IEEE, 2018.

[44] JJ Garcia-Luna-Aceves, Marc Mosko, Ignacio Solis, Rebecca Braynard, and Rumi Ghosh. Context-aware packet switching in ad hoc networks. In *2008 IEEE 19th International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–6. IEEE, 2008.

[45] Cesar Ghali, Gene Tsudik, and Ersin Uzun. Needle in a haystack: Mitigating content poisoning in named-data networking. 2014.

[46] Ali Ghodsi, Scott Shenker, Teemu Koponen, Ankit Singla, Barath Raghavan, and James Wilcox. Information-centric networking: seeing the forest

for the trees. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 1. ACM, 2011.

[47] C Hedrick. Rfc 1058: The routing information protocol (rip). *Internet Engineering Task Force (IETF) Request For Comments, http://ietf. org/rfc/rfc1058. txt*, 1988.

[48] 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.

[49] Pan Hui, Jon Crowcroft, and Eiko Yoneki. Bubble rap: Social-based forwarding in delay-tolerant networks. *Mobile Computing, IEEE Transactions on*, 10(11):1576–1589, 2011.

[50] Muhammad Mahmudul Islam, Ronald Pose, and Carlo Kopp. A hybrid qos routing strategy for suburban ad-hoc networks. In *The 11th IEEE International Conference on Networks, 2003. ICON2003.*, pages 225–230. IEEE, 2003.

[51] Natalie Ivanic, Brian Rivera, and Brian Adamson. Mobile ad hoc network emulation environment. In *Military Communications Conference, 2009. MILCOM 2009. IEEE*, pages 1–6. IEEE, 2009.

[52] Joshua Joy, Yu-Ting Yu, Mario Gerla, Samuel Wood, James Mathewson, and Mark-Oliver Stehr. Network coding for content-based intermittently connected emergency networks. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 123–126. ACM, 2013.

[53] Jussi Kangasharju, James Roberts, and Keith W Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25(4):376–383, 2002.

[54] Arseny Kapoulkine. pugixml – light-weight, simple and fast xml parser for c++ with xpath support. Online, accessed 13-October-2015.

[55] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11-16):1203–1213, 1999.

[56] Konstantinos Katsaros, George Xylomenos, and George C Polyzos. Multicache: An overlay architecture for information-centric networking. *Computer Networks*, 55(4):936–947, 2011.

[57] Vikas Kawadia, Niky Riga, Jeff Opper, and Dhananjay Sampath. Slinky: An adaptive protocol for content access in disruption-tolerant ad hoc networks. In *ACM MobiHoc 2011 International Workshop on Tactical Mobile Ad Hoc Networking*. Citeseer, 2011.

[58] Joud Khoury, Scott Nelson, Armando Caro, Vikas Kawadia, Dorene Ryder, and Tim Strayer. An efficient and expressive access control architecture for content-based networks. In *Military Communications Conference (MILCOM), 2014 IEEE*, pages 1034–1039. IEEE, 2014.

[59] T. Koponen, M. Chawla, B.G. Chun, A. Ermolinskiy, K.H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 181–192. ACM, 2007.

[60] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso. Manet simulation studies: the incredibles. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(4):50–61, 2005.

[61] Dirk Kutscher, Taekyoung Kwon, and Ignacio Solis. Information-Centric Networking 3 (Dagstuhl Seminar 14291). *Dagstuhl Reports*, 4(7):52–61, 2014.

[62] Nikolaos Laoutaris, Hao Che, and Ioannis Stavrakakis. The lcd interconnection of lru caches and its analysis. *Performance Evaluation*, 63:609–634, 2006.

[63] 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.

[64] 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.

[65] Jun Li, Hao Wu, Bin Liu, Jianyuan Lu, Yi Wang, Xin Wang, Yanyong Zhang, and Lijun Dong. Popularity-driven coordinated caching in named data networking. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 15–26. ACM, 2012.

[66] A. Lindgren, A. Doria, and O. Schelén. Probabilistic routing in intermittently connected networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 7(3):19–20, 2003.

[67] 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.

[68] James Mathewson. Fate - functional algebraic atomic evaluators. http:/github.com/jlmathew/Fate. Online.

[69] James Mathewson. Fate cache code and results for icnc2019. https://drive.google.com/file/d/1FYRE7SCiEWbJfWC1Icy04sp1uIhLNW2g/view?usp=shar Online.

[70] Zhourong Miao and Antonio Ortega. Scalable proxy caching of video under storage constraints. *IEEE journal on selected areas in communications*, 20(7):1315–1327, 2002.

[71] Scott Michel, Khoi Nguyen, Adam Rosenstein, Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive web caching: towards a new global caching architecture. *Computer Networks and ISDN systems*, 30(22):2169–2177, 1998.

[72] Zhongxing Ming, Mingwei Xu, and Dan Wang. Age-based cooperative caching in information-centric networks. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*, pages 268–273. IEEE, 2012.

[73] James Mathewson Maziar Barijough Ehsan Hemmati J.J. Garcia-Luna-Aceves Marc Mosko. Sconet : Simulator content networking. Online; accessed 13-October-2015.

[74] John Moy et al. Ospf version 2. 1998.

[75] Parc. Ccnx content centric networking project. Online; accessed 13-October-2015.

[76] Madhav S. Phadke. Taguchi methods using orthogonal arrays. Online; accessed 13-October-2015.

[77] Thomas Plagemann, Roberto Canonico, Jordi Domingo-Pascual, Carmen Guerrero, and Andreas Mauthe. Infrastructures for community networks. In *Content Delivery Networks*, pages 367–388. Springer, 2008.

[78] 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.

[79] Michael Rabinovich, Jeff Chase, and Syam Gadde. Not all hits are created equal: cooperative proxy caching over a wide-area network. *Computer Networks and ISDN Systems*, 30(22):2253–2259, 1998.

[80] Jing Ren, Kejie Lu, Fei Tang, Jin Wang, Jianping Wang, Sheng Wang, and Shucheng Liu. Caka: a novel cache-aware k-anycast routing scheme for publish/subscribe-based information-centric network. *International Journal of Communication Systems*, pages n/a–n/a, 2015.

[81] Pablo Rodriguez, Christian Spanner, and Ernst W Biersack. Web caching architectures: hierarchical and distributed caching. In *Proceedings of WCW*, volume 99, 1999.

[82] Giuseppe Rossini and Dario Rossi. Coupling caching and forwarding: Benefits, analysis, and implementation. In *Proceedings of the 1st international conference on Information-centric networking*, pages 127–136. ACM, 2014.

[83] Lorenzo Saino, Ioannis Psaras, and George Pavlou. Icarus: a caching simulator for information centric networking (icn). In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, pages 66–75. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014.

[84] Stefano Salsano, Nicola Blefari-Melazzi, Andrea Detti, Giacomo Morabito, and Luca Veltri. Information centric networking over sdn and openflow: Architectural aspects and experiments on the ofelia testbed. *Computer Networks*, 57(16):3207–3221, 2013.

[85] Stefan Saroiu, Krishna P Gummadi, Richard J Dunn, Steven D Gribble, and Henry M Levy. An analysis of internet content delivery systems. *ACM SIGOPS Operating Systems Review*, 36(SI):315–327, 2002.

[86] Mary R Schurgot, Cristina Comaniciu, and Katia Jaffres-Runser. Beyond traditional dtn routing: social networks for opportunistic communication. *arXiv preprint arXiv:1110.2480*, 2011.

[87] James Scott, Jon Crowcroft, Pan Hui, and Christophe Diot. Haggle: A networking architecture designed around mobile users. In *WONS 2006: Third Annual Conference on Wireless On-demand Network Systems and Services*, pages 78–86, 2006.

[88] D. Skeen. Vitria's publish-subscribe architecture. Online; accessed 13-October-2015.

[89] Ignacio Solis and J. J. Garcia-Luna-Aceves. Robust content dissemination in disrupted environments. In *Proceedings of the Third ACM Workshop on Challenged Networks*, CHANTS '08, pages 3–10, New York, NY, USA, 2008. ACM.

[90] M. Mosko I. Solis. Ccnx messages in tlv format. Online; accessed 13-October-2015.

[91] M. Mosko I. Solis. rfc8609. Online; accessed 17-Mar-2021.

[92] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 252–259. ACM, 2005.

[93] Thrasyvoulos Spyropoulos, Rao Naveed Rais, Thierry Turletti, Katia Obraczka, and Athanasios Vasilakos. Routing for disruption tolerant networks: taxonomy and design. *Wireless networks*, 16(8):2349–2370, 2010.

[94] Thrasyvoulos Spyropoulos, Thierry Turletti, and Katia Obraczka. Routing in delay-tolerant networks comprising heterogeneous node populations. *Mobile Computing, IEEE Transactions on*, 8(8):1132–1147, 2009.

[95] SRI. Edge networking with content-oriented declarative enhanced routing. Online; accessed 13-October-2015.

[96] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[97] 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.

[98] UCLA. Named data networking project. Online; accessed 13-October-2015.

[99] A. Vahdat, D. Becker, et al. Epidemic routing for partially connected ad hoc networks. Technical report, Technical Report CS-200006, Duke University, 2000.

[100] Markus Vahlenkamp. Information-centric networking. *Computer Science*, 2012.

[101] 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).

[102] Network Functions Virtualisation. Introductory white paper. In *SDN and OpenFlow World Congress, Darmstadt, Germany*, 2012.

[103] Duane Wessels and Kim Claffy. Icp and the squid web cache. *IEEE Journal on Selected Areas in Communications*, 16(3):345–357, 1998.

[104] Alec Wolman, M Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M Levy. On the scale and performance of cooperative web proxy caching. In *ACM SIGOPS Operating Systems Review*, volume 33, pages 16–31. ACM, 1999.

[105] S. Wood, J. Mathewson, J. Joy, M.-O. Stehr, Minyoung Kim, A. Gehani, M. Gerla, H. Sadjadpour, and J.J. Garcia-Luna-Aceves. Iceman: A system for efficient, robust and secure situational awareness at the network edge. In *Military Communications Conference, MILCOM 2013 - 2013 IEEE*, pages 1512–1517, Nov 2013.

[106] S. Wood, H. Sadjadpour, and J.J. Garcia-Luna-Aceves. Socratic: A social approach to network coding rate control. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 352–356, Dec 2014.

[107] Samuel Wood, James Mathewson, Joshua Joy, Mark-Oliver Stehr, Minyoung Kim, Ashish Gehani, Mario Gerla, Hamid Sadjadpour, and JJ Garcia-Luna-Aceves. Iceman: A practical architecture for situational awareness at the network edge. 2013.

[108] Ming Xie. P2p systems based on distributed hash table. *Computer Science, University of Ottawa*, pages 1–6, 2003.

[109] George Xylomenos, Christopher N Ververidis, Vasilios Siris, Nikos Fotiou, Christos Tsilopoulos, Xenofon Vasilakos, Konstantinos V Katsaros, George C Polyzos, et al. A survey of information-centric networking research. *Communications Surveys & Tutorials, IEEE*, 16(2):1024–1049, 2014.

[110] Wei You, Bertrand Mathieu, Patrick Truong, Jean-François Peltier, and Gwendal Simon. Dipit: A distributed bloom-filter based pit table for ccn nodes. In *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, pages 1–7. IEEE, 2012.

[111] Guoqiang Zhang, Yang Li, and Tao Lin. Caching in information centric networking: a survey. *Computer Networks*, 57(16):3128–3141, 2013.

[112] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, Patrick Crowley, Christos Papadopoulos, Lan Wang, Beichuan Zhang, et al. Named data networking. *ACM SIGCOMM Computer Communication Review*, 44(3):66–73, 2014.

[113] Zhourong Miao and A. Ortega. Scalable proxy caching of video under storage constraints. *IEEE Journal on Selected Areas in Communications*, 20(7):1315–1327, Sep. 2002.