Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Nejc Bizjak

Predpomnjenje v programsko definiranih brezžičnih omrežjih na podlagi predvidevanja konteksta

MAGISTRSKO DELO

ŠTUDIJSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Veljko Pejović

Ljubljana, 2016

University of Ljubljana
Faculty of computer and information science

Nejc Bizjak

Context prediction-based prefetching in software-defined wireless networks

MASTER'S THESIS

SECOND - CYCLE STUDY PROGRAMME
COMPUTER AND INFORMATION SCIENCE

The mentor: doc. dr. Veljko Pejović

Ljubljana, 2016

I want to thank my mentor, doc. dr. Veljko Pejović, for putting so much time and energy into this thesis. Thanks for your help, for your responsiveness and detailed review of my work.

Special thanks to Peter for being patient with my poor job contributions in the last month. You are the best boss!

Neža, hvala, ker si verjela vame. Brez tvoje podpore ne bi bilo tega magistrskega dela, in kar je še pomembneje, brez tebe ne bi pristal v stroki, v kateri zares uživam.

Zahvaljujem se tudi družini za vso podporo in prijateljem za nepozabna študentska leta.

Contents

Abbreviations

abbr.	**English**	**Slovene**
SDN	Software Defined Networking	Programsko definirano omrežje
LCA	Lowest Common Ancestor	Najbližji skupni prednik
CDN	Content Distribution Networks	Omrežje za dostavo vsebin
AP	Access Point	Točka dostopa
ISP	Internet Service Provider	Ponudnik internetnih storitev
DNS	Domain Name System	Sistem domenskih imen
DTMC	Discrete-time Markov chain	Markovske verige z diskretnim časom
LRU	Least Recently Used	Nazadnje dostopan
LFU	Least Frequently Used	Najredkeje dostopan
GDSF	Greedy-Dual-Size-Frequency	Požrešno-dvojna velikost in frekvenca

Povzetek

Naslov: Predpomnjenje v programsko definiranih brezžičnih omrežjih na podlagi predvidevanja konteksta

V tem magistrskem delu se posvetimo izboljšanju predpomnjenja za mobilne uporabnike v večjem brezžičnem omrežju študentskega kampusa. Najprej opišemo negativne učinke mobilnosti na razmere v omrežju in uporabniško izkušnjo. Predlagamo novo metodo, ki bo s pomočjo tehnologij programsko definiranih omrežij preusmerila uporabnikove zahtevke na optimalno postavljene omrežne predpomnilnike in s tem izboljšala uporabniško izkušnjo in razbremenila jedrne omrežne povezave. Naš prispevek je omrežna aplikacija, ki prek SDN krmilnika Ryu nadzoruje omrežne tokove. Aplikacija prejme podatke o premikih uporabnikov, izračuna kateri predpomnilniki v omrežju so najprimernejši in ustrezno preusmeri uporabnikove tokove na te predpomnilnike. Našo metodo smo testirali v simulatorju omrežja Mininet. Z uporabo realnih podatkov iz študentskega kampusa na Dartmouthu smo zasnovali več testnih scenarijev. Kot glavno karakteristiko uporabniške izkušnje smo merili zakasnitev od začetka do konca zahtevka. Kot indikator stanja omrežja smo merili promet preko jedrnih povezav. Naši eksperimenti so pokazali, da naša metoda dinamičnega preusmerjanja lahko pod določenimi pogoji prinaša opazne izboljšave v primerjavi s tradicionalnimi, statičnimi pristopi.

Ključne besede: Programsko definirana omrežja, predpomnjenje, brezžična omrežja, mobilnost uporabnikov.

Abstract

Title: Context prediction-based prefetching in software-defined wireless networks

In this master thesis we focus on improving in-network caching for mobile users in a large campus WiFi network. First we pinpoint the negative effects of mobility on network conditions and user experience. We propose a method leveraging SDN technology to redirect users' requests to optimally located cache servers, resulting in improved user experience and lowered burden on the backhaul and core network links. Our contribution is a network application that controls the flows in the network via an SDN controller. The application takes user's movement traces as an input, computes the optimal location of cache servers in the network and redirects user's flows accordingly. We tested our solution in a Mininet network simulator. We devised multiple scenarios using real-world movement traces from Dartmouth Campus. We measured requests delay as the main characteristic for user experience and data traffic over core and backhaul links as an indicator of network health. Our experiments show that for mobile users our dynamic redirection approach provides noticeable improvements over traditional, static caching methods.

Keywords: Software defined networking, caching, WiFi network, user mobility.

Razširjeni povzetek

V tem magistrskem delu se posvetimo izboljšanju predpomnjenja za mobilne uporabnike v večjem brezžičnem omrežju študentskega kampusa. V zadnjih letih se je število zmogljivih mobilnih naprav izrazito povečalo. Število naprav v IP omrežjih bo trikrat višje od svetovnega prebivalstva. Poleg števila pa se povečuje tudi zmogljivost teh naprav. Omrežni promet bo do leta 2019 predvidoma dosegel 22 GB na prebivalca [7].

Če bo omrežni promet naraščal s trenutnim trendom, bomo verjetno kmalu opazili več omrežnih zastojev in posledično slabšo uporabniško izkušnjo. V boju z naraščajočim prometom se zelo pogosto uporablja predpomnjenje. Predpomnilnik v spletni domeni predstavlja namensko napravo, ki shranjuje pogosto zahtevane objekte na trdem disku ali v delovnem pomnilniku. Z uporabo predpomnjenja dosežemo naslednje koristi [26]:

- **Zmanjšan promet na omrežnih povezavah**.
- **Zmanjšane zakasnitve za uporabnika**. Število skokov od odjemalca do predpomnilnika je manjše kot do izvornega strežnika. To postane še posebej pomembno, ko je izvorni strežnik zelo daleč stran ali pa se na teh skokih izgubi veliko paketov.
- **Zmanjšana obremenitev izvornega strežnika**. Na del odjemalčevih zahtevkov odgovori predpomnilnik, zato je izvornemu strežniku prihranjeno nekaj računskega bremena.

Tradicionalne metode predpomnjenja z večimi predpomnilniki ne upoštevajo morebitne spremembe lokacije uporabnikov, zato se lahko njihova učinkovitost zmanjša v določenih okoljih, kot so fakultetna in mestna brezžična omrežja. V našem delu obravnavamo predvsem to tematiko in predlagamo rešitev, ki izniči negativne posledice mobilnosti uporabnikov.

Naša rešitev temelji na novi paradigmi programsko definiranih omrežij (SDN), ki nam omogoča centralizirani, fleksibilnejši in enostavnejši nadzor nad omrežjem kot tradicionalne tehnologije. Naš prispevek je omrežna aplikacija, ki preko SDN krmilnika preusmerja določene uporabnikove omrežne tokove k skrbno izbranemu omrežnemu predpomnilniku. Aplikacija kot vhodni parameter prejme predvidene premike uporabnikov v omrežju, na podlagi katerih izračuna potrebne preusmeritve omrežnih tokov. Matematično lahko to opišemo kot problem k-sredin (angl. k-median) [1] in ga v našem delu rešujemo z uporabo vzvratno požrešnega algoritma (angl. Reverse-greedy) [6] in enostavnejšo metodo po principu najbližjega skupnega prednika.

V eksperimentih uporabimo realne podatke iz kampusa na univerzi v Dartmouthu, ki vsebujejo zapise vseh točk dostopa v povezavi z brezžičnimi karticami uporabnikov med letoma 2001 in 2003 [17]. Na podlagi teh podatkov lahko razberemo premike uporabnikov v omrežju in jih neposredno uporabimo v aplikaciji. Na istih podatkih s pomočjo hierarhičnega grupiranja generiramo omrežno topologijo, podobno tisti na univerzi v Darthmouthu. Simulacije izvajamo v simulatorju omrežja Mininet, v katerem namestimo tudi spletni strežnik. Na omrežna stikala namestimo predpomnilnike tipa Squid. Uporabimo 170 vozlišč, od katerih jih 40 predstavlja stavbe in 100 točke dostopa.

V prvem eksperimentu najprej potrdimo tezo o negativnih učinkih mobilnosti na predpomnjenje z uporabo večih predpomnilnikov. Eksperiment izvedemo na manjšem omrežju z dvema točkama dostopa in desetimi uporabniki na vsaki točki dostopa. Uporabniki dostopajo do večjega nabora vsebin in ko se predpomnilnik napolni, enega uporabnika premaknemo na drugo točko dostopa in merimo razmerje zadetkov v predpomnilniku za tega uporabnika. Ob premiku opazimo dvajset odstotni padec razmerja zadetkov, ki pa začne počasi naraščati proti prvotnim tridesetim odstotkom. To pojasnimo z dejstvom, da se je predpomnilnik, ki je odgovarjal na zahtevke tega uporabnika, v začetni fazi napolnil z delom njegovih datotek, po premiku pa drugi predpomnilnik ni vseboval teh datotek.

V naslednjem eksperimentu primerjamo našo rešitev s preprosto metodo predpomnjenja na prehodnem stikalu (gateway switch). Rezultati so pokazali opaznejšo zmanjšanje prometa v omrežju pri uporabi naše metode, kljub uporabi verjetnostih premikov uporabnikov s tehniko Markovskih verig. Prav tako pa se je zmanjšala zakasnitev, ki so jo občutili uporabniki v omrežju. Opazili smo, da je rezultat odvisen tudi od števila uporabljenih predpomnilnikov za vsakega uporabnika in značilnosti premikanja.

V stranskem eksperimentu smo primerjali različne tehnike menjave shranjenih objektov, ki jih ponuja Squid, kjer smo ugotovili, da se v našem primeru najbolje obnese tehnika, ki upošteva velikost in frekvenco zahtev shranjenih objektov (Greedy-Dual-Size-Frequency).

Zasnovali smo tudi eksperiment, kjer smo predpomnilnike postavili samo na stikala, ki predstavljajo stavbe. Našo metodo smo primerjali s predpomnenjem brez preusmeritev. Tudi tu se je naša metoda izkazala za boljšo, čeprav z manjšo razliko kot v drugem eksperimentu. Verjamemo pa, da bi naša metoda imela še opaznejše učinke, če bi uporabili novejše podatke, v katerih bi bili zastopani tudi uporabniki pametnih telefonov, ki so tudi mobilnejši.

Nazadnje preizkusimo obnašanje naše aplikacije v nepredvidljivih okoliščinah. V simulacijah uporabnike premikamo z naraščajočo mero naključnosti, tako da njihovo premikanje začne vedno bolj odstopati od predvidenega. Ugotovimo, da nepredvidljivost igra pomembno vlogo pri učinkovitosti naše aplikacije, saj pri večji nepredvidljivosti lahko naša aplikacija celo poveča promet v omrežju.

Področje omrežnega predpomnjenja je zelo široko in opazimo veliko možnosti za izboljšave. Rešitev bi bilo potrebno predvsem preizkusiti v različnih okoljih z različnimi parametri, tako da bi v celoti spoznali prednosti in slabosti predlagane rešitve. Zagotovo je veliko prostora za napredek tudi v samem algoritmu. Največja izboljšava bi bila uporaba metode za pametno izbiro števila uporabljenih predpomnilnikov za vsakega uporabnika, saj so uporabniki v eksperimentih imeli zelo različne vzorce premikanja. V aplikaciji smo uporabili požrešni algoritem, ki pa vrne samo približno najboljšo rešitev. Sklepamo, da bi z uporabo natančnejšega algoritma lahko dosegli nekoliko boljše rezultate.

Chapter 1

Introduction

Over the last few years the number of powerful mobile devices has increased drastically. The number of devices connected to IP networks will be three times as high as the global population in 2019 [7]. Besides quantity, capabilities of those devices are also increasing. By 2019 IP traffic will reach 22 GB per capita, up from 8 GB per capita in 2014 [7].

If the trend of demand outpacing the capacity growth continues, we will probably soon see more frequent network congestions and worse quality of experience. Caching is the main technique used to unburden the network links and improve response times [26]. A web cache is a dedicated machine that stores frequently accessed objects in RAM and on the hard disk. Caching in the internet domain provides us with the following benefits:

- **Reduced traffic on internet links**.
- **Reduced delay for the user**. Hop count from client to proxy (cache server) is lower than hop count to original server. This becomes especially important when original server is far away (another continent) or there is high packet loss on these hops.
- **Reduced server load**. Part of the requests are served by proxies so original server sees this as lower amount of requests which results in lower computational burden on the server.

In the internet domain we can distinguish three most commonly used cache types. **Browser cache** is used on the client side. It stores content that a user most commonly accesses via a Web browser. It provides the highest gains in terms of reducing the delay, but it also lowers the power consumption on the client, network and server side since no network activity is needed when content is served from the local cache. The trade-off here is that storage on client devices can be expensive and limited, especially on smartphones.

Network cache servers (proxies) (Figure 1.1) are used in networks on all levels, private networks, ISP networks, etc. These cache servers are usually dedicated machines that serves a group of users. They are usually deployed for scalability reasons, because they reduce the network load, especially on the backhaul links. A network that incorporates cache servers can usually serve more users than a network without them. If we give an example from the cost perspective, an organization can have a cheaper, lower capacity backhaul link, if it has a cache server deployed in the network, because this link will not be as burdened as it would be if cache server would not be used.

Special example is **Content Distribution Network** (CDN), which is a globally distributed network of cache server that serves larger user population (Figure 1.2). The main difference besides scale of deployment is the business model since content owners pays to CDN provider to serve their content.

Reverse proxy cache server (Figure 1.1). Normal or *forward* proxy serves a group of clients by requesting and caching the content from the source server on their behalf. Reverse proxy is deployed "in front" of the source servers and answers requests on behalf of these servers. This is often used to lower the computational burden on source servers.

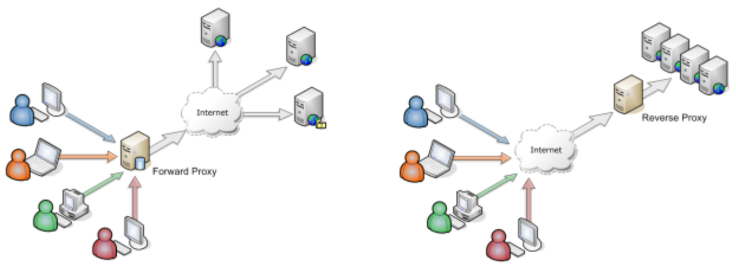


Figure 1.1: Forward vs. Reverse proxy. Source: `http://http://community.brocade.com/t5/vADC-Docs/Using-Stingray-Traffic-Manager-as-a-Forward-Proxy/ta-p/73721`

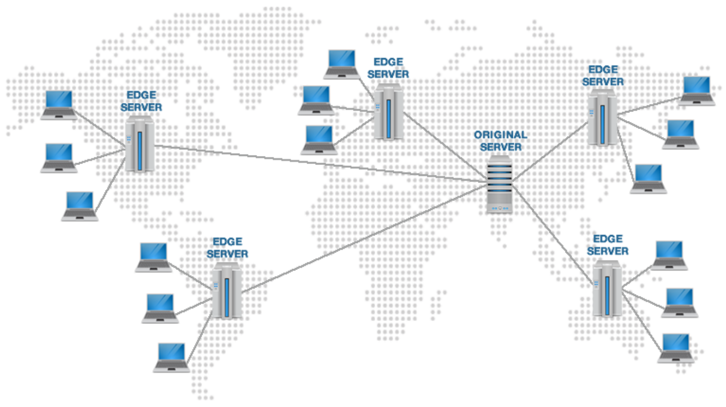


Figure 1.2: Content Distribution Network. Source: `http://www.cdnreviews.com`

Traditional caching techniques that use multiple cache servers do not account for the changing location of the users, so they may perform worse in certain environments like metropolitan WiFi or campus WiFi networks, where users are more mobile. Some techniques, like content relocation, were proposed, but they are not very efficient or are very difficult to implement and thus are not portable [19]. In this work we put a strong emphasis on

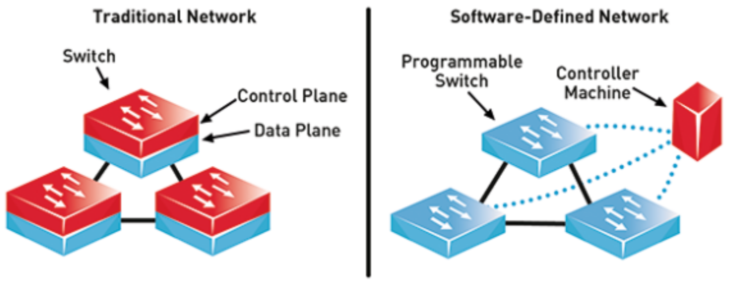


Figure 1.3: Traditional vs Software-Defined network [3]

effect of user mobility on caching and design a solution to mitigate those effects.

The rise of Software-defined networking (SDN) gives us new opportunities to deal with these challenges. This new paradigm gives us centralized and programmable control over network. To express the desired high-level policies in traditional networks, network operators need to configure each individual network device separately using low-level and often vendor-specific commands. SDN moves the control logic from network devices to centralized network controller and simplifies and abstracts network configurations (Figure 1.3).

The network is programmable through software applications running on top of the controller that interacts with the underlying data plane devices (Figure 1.4). Additionally, destination-based packet forwarding is replaced with broader and more flexible flow-based forwarding. A flow is defined by a set of packet field values and actions applied to the packets of this flow [18].

We propose a novel approach that would adapt the caching to the mobility of the user. We create a network application that can modify network flows based on user's movement pattern in the WiFi network to redirect the user to specific cache servers that would serve the user best. Our intention is to create portable solution, independent of network topology and movement traces. Our solution is tested on real-world mobility traces from Dartmouth campus data trace [17] and on topology generated from access points from

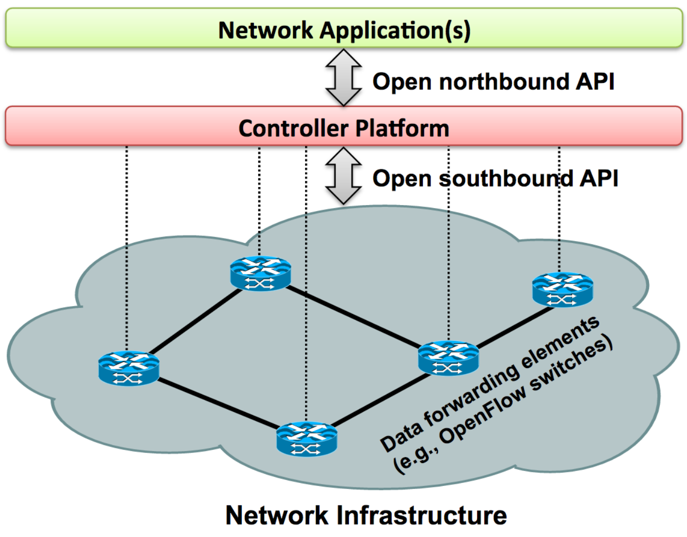


Figure 1.4: Simplified view of an SDN architecture. Source: `https://www.grotto-networking.com/BBSDNOverview.html`

same dataset. We also outline the limitations of proposed approach and suggest possible future improvements and directions for future research.

Chapter 2

Related Work

User experience in Web browsing is the sum of network conditions, server performance and various client-side effects like hardware performance, local storage, etc. A lot of research was done on client-side to improve user experience. This is often complemented with research on mobility predictions. Interesting work was done in [24], where authors used mobility and throughput predictions to proactively cache video content on access points the user encountered along the way. Orchestration was done by specially configured mobile device. Although results are good, the solution itself does not scale very well, because special modifications are needed on the client-side with the exact knowledge of a network topology. Our solution do not need any client-side modifications so we avoid these problems completely.

In another research [9] mobility prediction algorithms have been used to control downloading schedule and buffering on the Android smartphone trying to minimize segment lateness and maximize video quality by predicting data rates in user movement through the network. Both solutions and ours rely on mobility prediction and result in improvements to user experience but each in very different way. This shows that there is a lot of potential to improve user experience just by exploiting mobility predictions on client and network side solutions.

Network caching has always been popular research topic, because it provides substantial gains in terms of lowering the network traffic and request latency. This problem domain has a lot of different tunable parameters and different use cases and many aspects have been extensively researched. A lot of work has been done on cache replacements schemes [27]. These policies determine which piece of content will be removed from local storage when new object needs to be stored. By nature, cache operates by using up the dedicated disk space and cached objects are continuously being evicted and replaced with the new ones. That is why replacement policies have such an important role. There exists five groups of policies, each containing numerous variations:

- **Recency-based**. Eviction based on time of last access of cached objects.
- **Frequency-based**. Eviction is made based on how many times cached objects were accessed.
- **Size based**. Eviction is made based on size of cached objects.
- **Function based**. Eviction is specified by arbitrary function which may take into account various parameters of cached objects like recency, frequency, size, etc. One example of this function is Greedy-Dual-Size-Frequency algorithm, which takes into account all three parameters explained above.
- **Randomized policies**. Policies that have probabilistic behaviour, but can still take into account any of the parameters explained above.

Another problem that received a lot of attention was the placement of cache servers in the networks. There exists special framework that determines the optimal placement of cache servers [20]. In our thesis we also deal with this problem, but from different perspective. We do not choose where to place cache servers, but we select to which already placed cache server should user's request be redirected. Also related to our work are studies of effect of mobility on caching. Interesting approach to support user mobility was proposed in [19], where the authors developed two algorithms to move the cached content so it remains close to moving clients. Although they reported improvement over existing path-prediction based approaches, overhead was still quite big. We pursue the similar goal of supporting user mobility, but instead of moving the content around the network, we redirect users' requests to different cache servers instead to avoid the overhead reported in the mentioned study.

We found interesting and relevant solution called EdgeBuffer [28]. This platform is part of MobileFirst project, which tries to address the drawbacks of traditional networks originally built for static devices. They propose using wireless access points as cache nodes. They take an interesting approach with a prefetch buffer that takes into account individual access patterns of the users and also general popularity of the content. Although their experiments showed significant improvements over traditional caching methods, we think such solutions are too difficult to implement in regular networks because they need special infrastructure and configuration. We try to overcome this drawback by building our solution upon SDN, which makes the implementation easier and no special infrastructure is needed, except for the switches that support OpenFlow protocol.

Emergence of a new paradigm called Software-defined networking (SDN) has introduced new possibilities for inovations in network management [15]. Control logic over whole network is centralized and programmable while network devices perform only packet forwarding. This gives us a platform on which new ideas can be introduced in the network through a high-level software program as opposed to using a fixed set of commands in proprietary network routers.

In LTE network domain, SDN based caching was also researched [16]. Although the research is from different domain, the authors are dealing with similar problems like mobility and determining the best caching decision. They use the technique of content relocation. We decided to use the opposite method of request redirection. So instead of moving the content around, which is an expensive operation, we reroute the request to a target cache server to get the similar effect but faster and with much less network traffic.

OpenCache is a good example of a novel solution that was built upon SDN technology. It is experimental caching platform developed to provide more dynamic caching solution [2]. Authors pursue the goal of reducing the duplicate deliveries of identical content to lower the transit costs and improving the user experience. Their solution presents a cheaper alternative to Content Distribution Network (CDN) and is more suitable for deployment in smaller networks and last-mile environments. The solution has built-in function for creating the nodes, moving, fetching and seeding the content so we think that their platform presents numerous new opportunities for improving cache related problems. Their solution is also tested on a large-scale testbed deployed across Europe. We also tried to use this platform in our experiments, but due to lack of technical documentation we abandoned this approach. Their solution operates directly with custom cache servers but ours leaves the caching to more established solutions like Squid proxy and focuses only on network flow modifications. They provide more general solution while we tackle very specific problem of user mobility in the network.

Studying the related work on this field, especially SDN related experiments, gave us a lot of valuable insights and ideas to use in our work. We think that user mobility does introduce new challenges on all sides in Web domain. We also think that these challenges can be tackled by incorporating mobility predictions in decision making process of developed solutions.

Chapter 3

Problem Statement

3.1 Scenario

Imagine a large campus network with 160 buildings, over 600 access points and 3000 network users, which are served by one gateway to internet and multiple cache servers. For most of the time the users are static. Students are in the dorms or classes, professors and other employees are in their offices and are not very mobile. In these cases network environment is static and stable. Cache servers are serving fixed group of users with the content that is most frequently requested by this specific group.

Problem arises when users become mobile and go to area that is covered by a different cache server. There are many transitions like this in a normal day. Students are moving between dormitories and academic buildings, professors are moving between cabinets and classrooms, everyone goes to cafeteria at one point in a day, etc. During transitions like these, user's requests are going to different cache server, which probably stores very different content than the cache server the user was served from before, so there is a higher chance of *Cache Miss* events. This increases the delay experienced by the user, because content has to be retrieved from the original source that is probably many hops away. Besides that, this new content will cause cache to evict some of the existing files to free the space for storage of a new object.

This is expected behavior, but it decreases overall cache performance if user will stay at this location only for a small period of time.
Consequences:

- Average round trip time of requests is longer than in static cases due to the fact that requests go to source server many hops away.
- Network traffic on backhaul and core links is increased so there is higher probability of congestion.
- Cache servers are slowly repopulated with the content of a new group of users.

3.2 Problem formulation

To reduce the negative effects of mobility on the network conditions we must redirect the user's requests to the cache server that would serve him with the highest hit rate. In this thesis we assume that a large network is organized in a tree structure so this problem can be translated into finding k-median in a tree (undirected acyclic graph). In the k-median problem we are given a graph G in which each node u has a non-negative weight w_u and each edge (u,v) has a non-negative length d_{uv}. We extend this notation to arbitrary u,v $\in$ G, so that d_{uv} is the minimum distance between u,v in G. Our goal is to find a set F of k vertices that minimizes $cost_F(G) = \sum_{x \in G} \min_{u \in F} w_x d_{xu}$. The optimal cost of G with k cache servers is $cost_k(G) = \min \{cost_F(G) : |F| \leq k\}$ [1].

In our problem domain u represents access points (leaf nodes) from which individual user with id i made w_{ui} requests. F is a set of k cache servers that serves this user. It costs d_{xui} to serve a user request at access point x from cache server located at u. We wish to redirect user to k cache servers in G so that overall serving cost is minimized. This optimal set of k cache servers is called k-median of G for a specific movement pattern of a user [1].

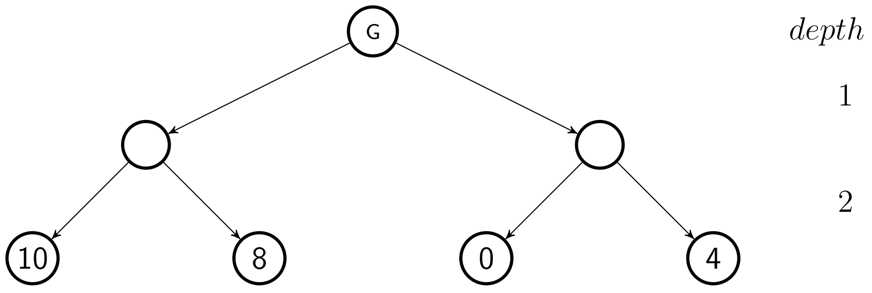


Figure 3.1: Example tree representation of the network with weighted leafs that represents the number of user's request on each access point.

Kariv and Hakimi [14] proved that this problem is NP-hard in the case of general graphs, and proposed an $\mathcal{O}(n^2k^2)$ algorithm for undirected trees, where n is the number of tree nodes and k is the number of cache servers.

Our network is represented by tree where root represents gateway to internet (node marked with G), leaf represents an WiFi access point and intermediary nodes represents networks switches. Each leaf node carries a value which represents a number of processed user requests. For the sake of the demonstration, we assume that cache servers are placed on all network nodes. In a selected time slot user has made ten, eight and four requests on first (from the left), second and fourth access point respectively. In the next chapter we will describe two algorithms we implemented for solving *k-median* problem described in this section. All of the described algorithms use the topology displayed in Figure 3.1 as a starting point.

Chapter 4

Proposed Solution

4.1 Single median

We start with the most simple instance of k-median problem, single median (one selected cache server per user). This means that we can forward user's requests to one designated cache server in the network. We must now compute which cache server would be the most appropriate for user's distribution of requests.

We devised simple cost function that we will use to evaluate possible cache server selections (Equation 4.1). With our cost function we wanted to take into account the number of issued requests and also penalize the use of core network links, because by selecting cache servers closer to the edge we can significantly reduce the in-network traffic on core and intermediary links. Closer the link is to the gateway (lower depth), higher the cost. A cost of cache server being selected to serve a specific user is the sum of all link costs from each AP to designated cache server.

$$\boxed{Cost(user, cache) = \sum_{i=1}^{|APs|} \sum_{d=depth(AP[i])}^{depth(cache)} \frac{numberOfRequests(user, AP[i])}{d}} \tag{4.1}$$

We start by computing the Lowest Common Ancestor (LCA) of access points with number of requests higher than zero (Algorithm 1). LCA algorithm first gets the *Paths* for each AP. Path is a list of nodes between AP and the gateway ordered by increasing depth. First element on all paths is always the gateway (root) node. Root node is also our first candidate for LCA. We compute the cost of the candidate with cost function 4.1 and compare it with the costs of candidate's children. If all children have higher cost than the parent then the algorithm returns the parent as the LCA, otherwise the child with the lowest cost is promoted to be the next candidate and the process repeats. Computed LCA is the first candidate for optimal solution and is sent to Algorithm 2 as an argument x. If we would select cache server anywhere higher than on LCA, we would always have higher hop count and thus higher delays for requests without any additional benefits, as there are no requests in other parts of the network that might benefit from this higher placement of cache server.

Algorithm 1 Lowest Common Ancestor

```
1: for all $\{AP_i \in APs : numberOfRequests(AP_i) > 0\}$ do
2:     # $Paths[i] = [root, node\_at\_depth\_1, node\_at\_depth\_2, \ldots, node\_i]$
3:     $Paths[i] \leftarrow path(G, AP_i)$
4:     $lastI \leftarrow i$
5: end for
6: $minDepth \leftarrow min_i(|Paths[i]|)$
7: $Depth \leftarrow 0$
8: $CurrentLca \leftarrow Root$
9: while $Depth \leq minDepth$ do
10:     # First LcaCandidates will be one of the CurrentLca's children
11:     $LcaCandidate \leftarrow Paths[lastI][Depth]$
12:     for all $Path \in Paths$ do
13:         if $Path[Depth] \neq LcaCandidate$ then
14:             return $CurrentLca$
15:         end if
16:     end for
17:     $CurrentLca \leftarrow LcaCandidate$
18:     $Depth \leftarrow Depth + 1$
19: end while
20: return $CurrentLca$
```

We compute the cost of selecting the cache server on LCA with equation 4.1. Then we compute the cost of caching on all of the LCA's children. If children's costs are higher than the current candidate's cost, we declare current candidate as optimal solution, otherwise we recursively continue with this process on the child with the lowest cost.

LCA is chosen as the first cache candidate and cost is computed with equation 4.1. In the first step, the algorithm will return the following result displayed in figure 4.2. LCA is chosen as a cache server candidate. Cost for selecting this cache server is the sum of all link costs $5 + 4 + 2 + 18 + 4 = 33$ (Figure 4.2). Cost is computed for each link between AP and cache server, where number of requests are divided by link depth factor. Higher the links are, higher the cost will be. As we stated before, one of our goals is also to unburden core links.

In the next step we compute the cost for next cache candidates (both children of the LCA node). Left child has a cost of 19 (Figure 4.2) and the right child has a cost of 47 (Figure 4.3). Previously selected node is marked red for easier demonstration. Left child has smaller cost than the parent and the right child, so we choose it as current best solution and continue with the algorithm from this node down.

Again, we compute the costs of children of current solution. Left child has a cost of 21 and the right child has a cost of 22. Because both children have higher cost than the parent, algorithm stops and the current best solution is returned as the final result (Figure 4.2).

Algorithm 2 1-median algorithm

```
 1: # Initialized with $x \leftarrow LCA$
 2: procedure SingleMedian($G, x$)
 3:     $Candidates \leftarrow children(G, x)$
 4:     $i \leftarrow argmin_i(cost(G, Candidates[i]))$
 5:     if $cost(G, Candidates[i]) < cost(G, x)$ then
 6:         return SingleMedian($G, Candidates[i]$)
 7:     else
 8:         return $x$
 9:     end if
10: end procedure
```

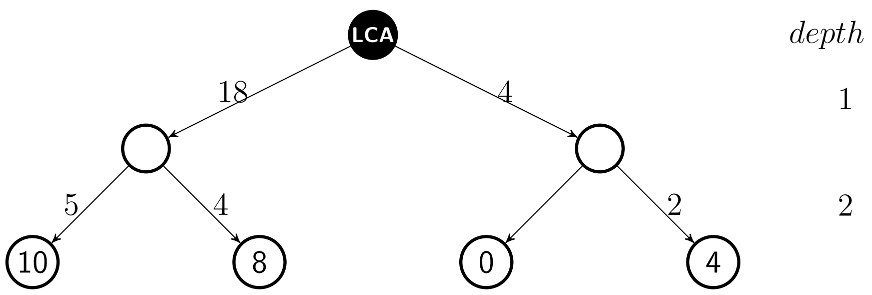


Figure 4.1: Initialization phase - LCA is cache candidate

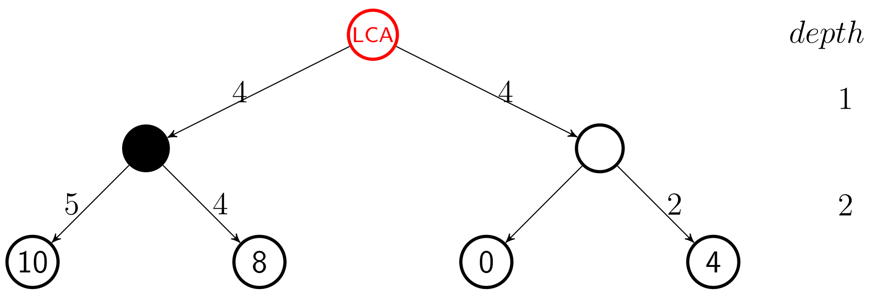


Figure 4.2: First temporary state in Single median algorithm

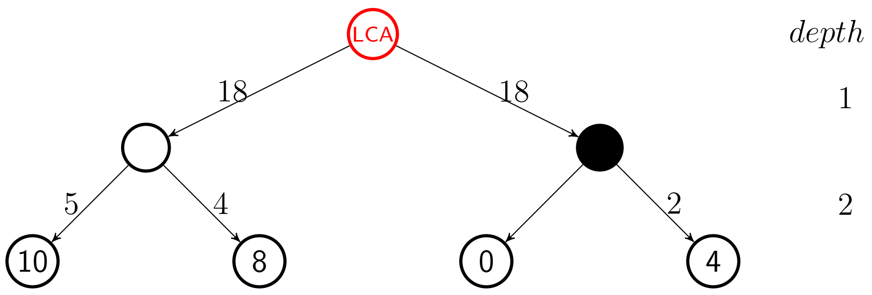


Figure 4.3: Second temporary state in Single Median algorithm

4.2 Reverse Greedy Algorithm

To solve k-median problem where $k > 1$, we implement Reverse Greedy Algorithm. General algorithm starts with all cache servers selected. In each iteration a cache server is deselected in such way that the overall cost increases the least. Algorithm stops when k cache servers remain selected in the network. Time complexity of this algorithm is $\mathcal{O}(\log n)$, where n is the number of all nodes in the network [6].

Algorithm 3 Reverse Greedy Algorithm [6].

```
1: # $F$ is a set of all cache servers in the network
2: # $u$ is a user for which we are selecting cache servers
3: procedure ReverseGreedy($F, u$)
4:     $S \leftarrow F$
5:     while $|S| > k$ do
6:         # $f$ is a cache server that introduces the least cost
7:         $f \leftarrow argmin_i(Cost(u, S - \{i\}))$
8:         $S \leftarrow S - \{f\}$
9:     end while
10:    return S
11: end procedure
```

Our problem domain has this specific characteristic that all requests are issued from leaf nodes, so first we need to do few adaptations to the original algorithm. Instead of starting with all cache servers selected for caching, we select only cache servers on access points that were visited by the user (number of requests is greater than zero). Request are always forwarded from AP to the closest selected cache server, so cache servers on intermediate and core nodes would not be used in this case. For this adaptation we define the following terms:

- **Path**. Path is a list of nodes between AP and the root node.
- **Clear path**. Path where only one cache server is selected.
- **Collision**. State where there are two selected cache servers on the same path.

Our algorithm in each iteration performs one of two actions on all selected cache servers:

- **Move up**. Select the parent cache server and deselect the target one, This action is performed when path is clear. Number of total selected cache servers remains the same.

- **Merge**. When *collision* occurs we deselect one of the selected cache servers to clear the path (removes the collision). With this action we reduce the total number of selected cache servers ($K = K - 1$).

Initialization phase always produces state with all clear paths. In each iteration we perform one of the actions on all selected cache servers. Each action produces new state. For each state we compute the cost with equation 4.2. Equation is basically the same as in Single median algorithm (Equation 4.1), except here we are dealing with multiple cache servers, so we are computing the cost from AP to it's closest cache server $cache_{closest}$. Equation is user specific because it computes only requests made by specific user.

$$\boxed{Cost(user, caches) = \sum_{i=1}^{|APs|} \sum_{d=depth(AP[i])}^{depth(cache_{closest})} \frac{numberOfRequests(user, AP[i])}{d}} \tag{4.2}$$

Our initialization step is displayed in Figure 4.4. If $k = 3$ then this would also be the end result, but to demonstrate the process we compute the result for $k = 2$. We can observe that all cache servers (black nodes) have clear paths, which means that in next iteration there will be three separate "Move up" actions resulting in three temporary states as shown in Figures 4.5, 4.6 and 4.7. Node on which an action was performed is marked red for easier demonstration.

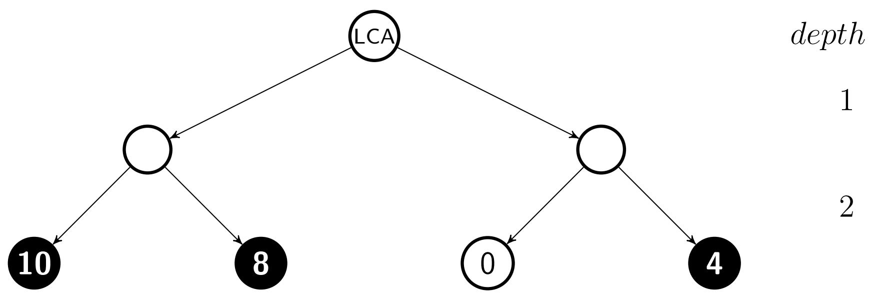


Figure 4.4: Reverse Greedy Algorithm - initialization phase

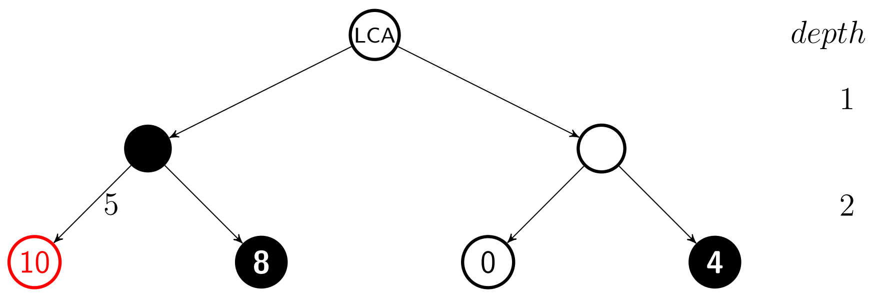


Figure 4.5: First iteration, first temporary state - “Move up” on first cache server

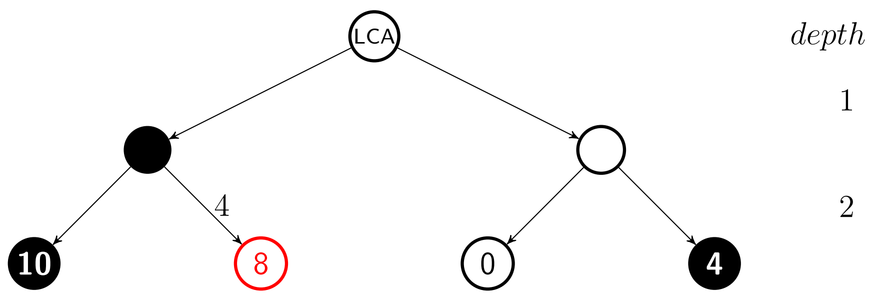


Figure 4.6: First iteration, second temporary state - “Move up” action on second cache server

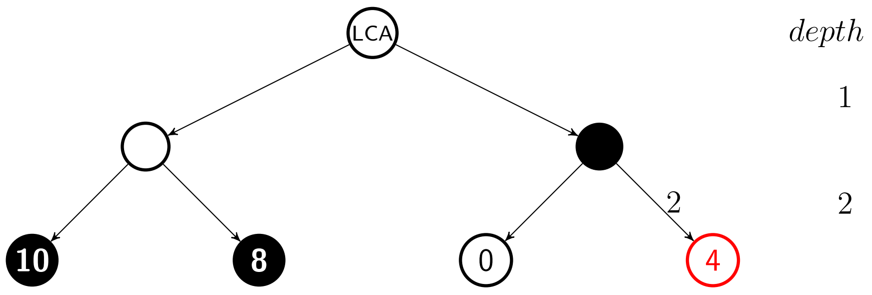


Figure 4.7: First iteration, third temp. state - “Moving up” third cache server

By definition of the Reverse Greedy Algorithm we move towards **least increasing** cost. We start with zero cost and after each iteration we decide which action will increase our total cost the least. In our example, we can observe that “Move up” action on the fourth cache server will increase the cost the least. Third temporary state becomes current state and we continue with the algorithm from this state forward.

We still do not have any collisions on the paths so next iteration will again perform three “Move up” actions. Costs are 7, 6 and 6 for each temporary state respectively. This time, second and third temporary state increase the total cost the least. In such cases, where multiple states have the same cost, we choose the one where requests have traveled the least distance. If candidates are still tied after this criteria, we give priority to candidate on higher depth. We decided for this criteria because we do not want to burden the core links.

In third iteration, we have the first collision. We have two cache servers on the first path. This means that the first and second actions will be “Merge” actions. Third state will be the same as in second iteration (Figure 4.8).

Three temporary states have the costs 11, 10 and 10. We can see that merging down would be slightly less costly than merging up. We must now decide between second and third state. Here we have a trade-off between burdening core link (third state) and increasing hop count for larger number of requests and increasing traffic more (second state). We decide to always

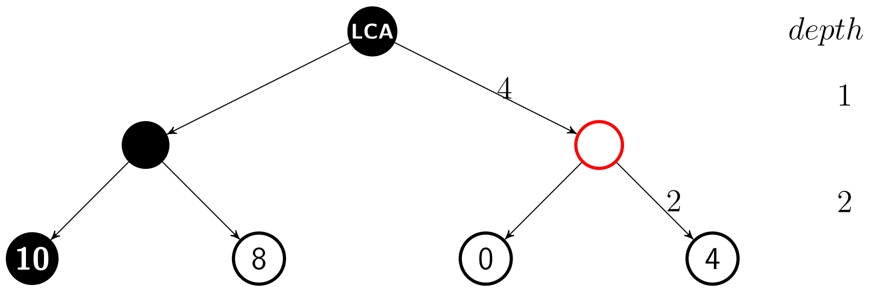


Figure 4.8: Second iteration, third temporary state

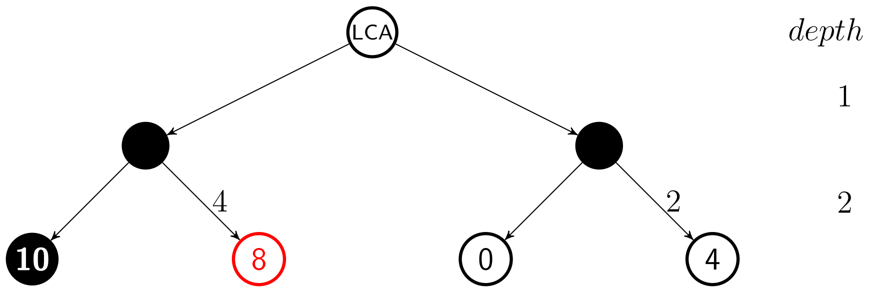


Figure 4.9: Second iteration, final state

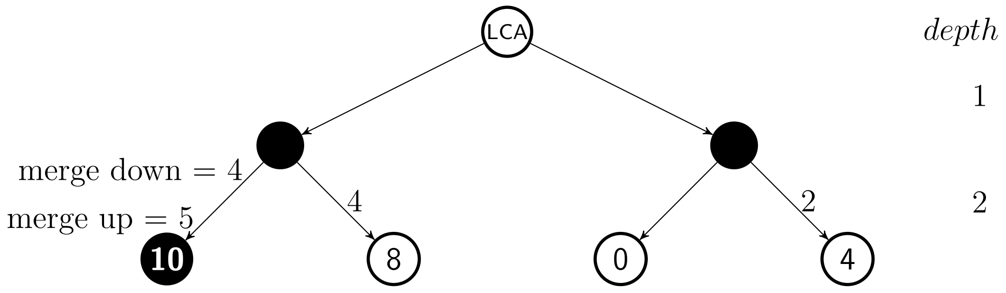


Figure 4.10: Third iteration, first and second state - “Merging” first two nodes

give priority to “Merge” actions, because despite possible worse short term results, we are decreasing the k, which brings us closer to the end solution and overall better cost. With this merge we also reach our exit condition $k = 2$. End results looks like tree in figure 4.11. We instantly see that this is not optimal solution, because second cache server could be placed at the AP directly thus avoiding one hop.

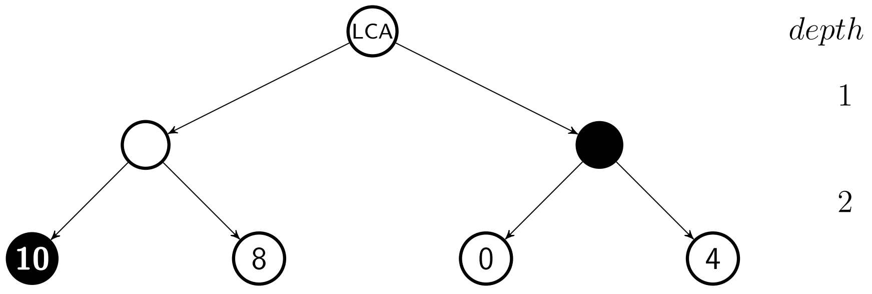


Figure 4.11: End result of Reverse Greedy Algorithm

In the worst case our algorithm performs $(n-k)*2-1$ “Move up” actions and $(n-k)$ “Merge” actions, where k is the number of selected cache servers per user and n is the number of nodes that have cache servers installed. Worst case is when all APs have greater than zero requests and $k = 1$. Only the actions that are actually executed and not discarded are taken into account. These observations are true for network in shape of binary tree, although in trees where node can have more than two children number of “Move up” actions is lower. If we include the actions that were discarded during the process we get $-1 + \sum_{i=n}^{k+1}(i-k)*2$ ”Move up” actions and $\sum_{i=n}^{k+1} i-k$ “Merge” actions. Findings presented in this paragraph apply for computation made for single user.

After merge, we do not perform any more actions on this node, only on target node. If we combine this with the fact that all paths are leading to the root, we can see that all APs can be merged with one another. Since the algorithm does incremental steps (only one action per iteration is applied), it will always converge to k with known worst case number of steps, as we show above.

4.3 Optimization

We saw in previous example (Fig. 4.11) that Reverse Greedy Algorithm did not return the optimal solution. Right cache server is clearly not in local minimum, because it would cost less to have cache server selected on the access point itself. To move the cache server in local minimum we apply same procedure as with Single Median algorithm 2. We try to select the children of currently selected cache servers to see if the costs decreases. Since left cache server is already at the edge, we can only move the right one. Moving it to the left children would increase the cost by 2, moving it to the right children would decrease the cost by 2.

This simple optimization puts our cache servers in local minimum, but it still does not guarantee to be optimal solution as seen in Figure 4.12.

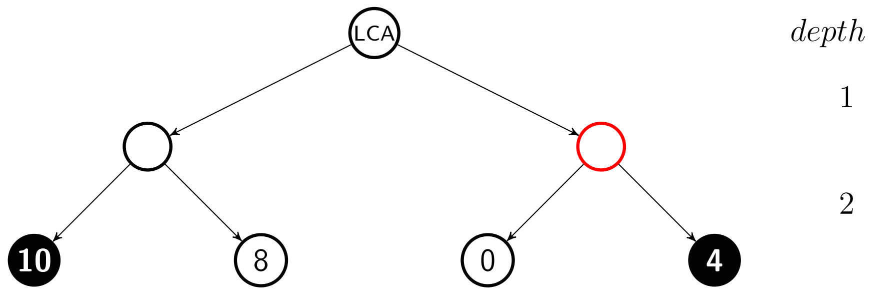


Figure 4.12: End result after optimization

Chapter 5

Experiments

5.1 Networking Technologies

The main enabler of our proposed solution is a new network paradigm called Software Defined Networking (SDN) that allows us to dynamically control network flows. SDN is a dynamic, manageable and adaptable architecture that decouples the network control and forwarding functions [18]. This decoupling and abstraction of physical infrastructure makes the network programmable and vendor-neutral [15]. It eliminates the complexity, issues with vendor dependency and scaling problems of traditional networks [15].
Network abstraction and decoupling is done with OpenFlow protocol [12]. This is also the basis for building SDN applications. In Figure 5.1 we can see an overview of SDN controller and OpenFlow switch in action. When switch receives a packet, it checks if there are any matching rules in the flow table. Matching can be done on various packet fields like source and destination IP and MAC address, VLAN id, input port, etc. If no match is found, switch sends the inquiry with packet header to the controller via the OpenFlow protocol. Controller examines the header and answers the inquiry with the rule that is inserted into flow table of this switch. What kind of rule will be inserted depends on the applications that are running on the controller. Packet headers can be processed by some Firewall application,

Quality of Service application, etc. When switch receives the rule from the controller, it executes the action defined in the rule. In this example, *Packet out* action was specified, so switch will forward the packet through the port specified in the rule.

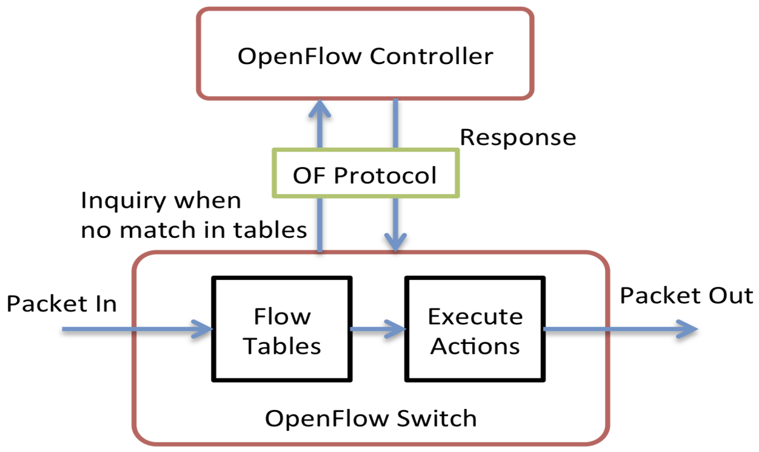


Figure 5.1: OpenFlow overview. Source: `https://s3f.iti.illinois.edu/usrman/openflow.html`

5.1.1 SDN controller

The concept of SDN is becoming mature as can be seen with many open source and commercial SDN platform choices. Popular open-source SDN controllers include Ryu, OpenDaylight Open vSwitch, and Trema, to name a few. Large vendors also offer commercial solutions like HP Virtual Application Networks, Cisco Application Policy Infrastructure Controller, NEC ProgrammableFlow, and VMware NSX Controller.

After testing the largest project, Java-based OpenDaylight, we decided to use *Ryu controller*, because OpenDaylight is overly complex for a proof-of-concept experiments we are set to perform in this thesis. Ryu is an easy-to-use SDN controller, has fast startup time and follows the standard SDN architecture (Figure 5.1.1). The controller has a REST client so it can receive commands via network from 3^{rd} party applications.

Southbound layer supports different protocols like OpenFlow, Network Configuration Protocol (NETConfig), Open vSwitch Database (OVSDB), etc. It also provides some built-in features like Topology discovery and Firewall, although in this aspect is OpenDaylight or some other commercial controller far more superior.

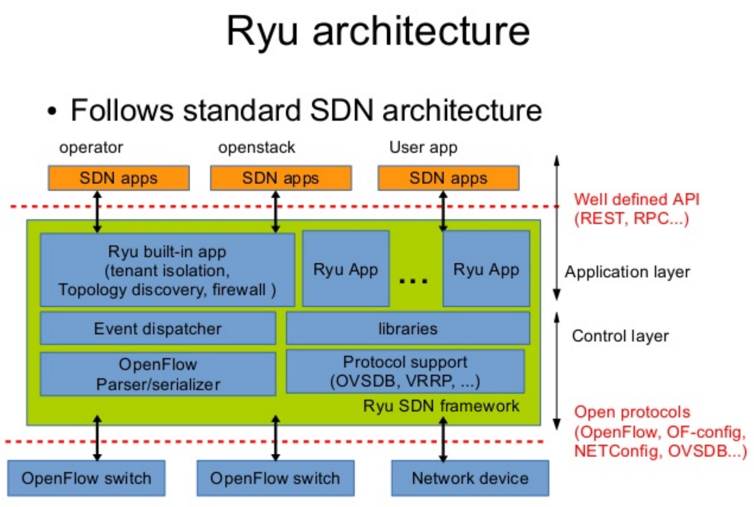
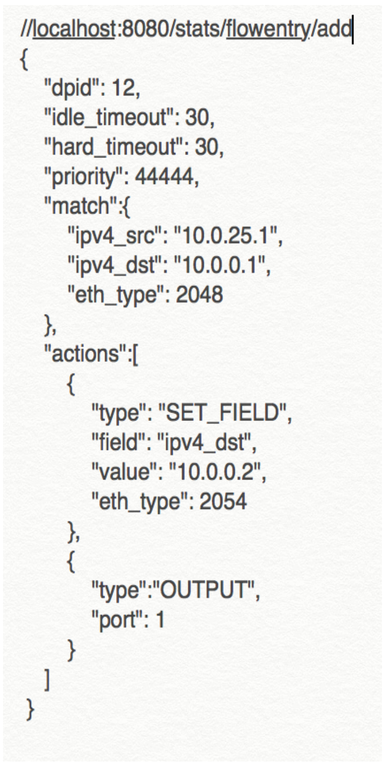

Ryu architecture
• Follows standard SDN architecture
operator
openstack
User app
SDN apps
SDN apps
SDN apps
Well defined API (REST, RPC...)
Ryu built-in app (tenant isolation, Topology discovery, firewall)
Ryu App
...
Ryu App
Application layer
Event dispatcher
libraries
Control layer
OpenFlow Parser/serializer
Protocol support (OVSDB, VRRP, ...)
Ryu SDN framework
Open protocols (OpenFlow, OF-config, NETConfig, OVSDB...)
OpenFlow switch
OpenFlow switch
Network device

Figure 5.2: Ryu architecture diagram. Source: `http://www.slideshare.net/yamahata/ryu-sdnframeworkupload`

We had two implementation options. We could build a Ryu module that would be loaded in Ryu controller. Benefit of this approach is that is more reactive since it can act on flows that controllers receives. But since we do not need that kind of power, we decided for the second option, a standalone application that communicates with the controller with the help of Ofctl-Rest module. We chose the second option also because it was easier to connect to data processing script and also use it directly in Mininet simulations.

Using the above approach our application sends commands to Ryu's REST module in a human-readable JSON format 5.1.1. Figure 5.1.1 depicts an example command sent to the Ofctl-Rest module of our Ryu controller, which parses the command and sends it via OpenFlow protocol to switch with Datapath ID (dpid) of 12, where it is inserted to flow table. This specific command will insert the rule that will change the destination IP of packets that are originating from IP *10.0.25.1* and are headed to *10.0.0.1*. These packets will also be forwarded through port 1 of this switch.



```
//localhost:8080/stats/flowentry/add
{
    "dpid": 12,
    "idle_timeout": 30,
    "hard_timeout": 30,
    "priority": 44444,
    "match":{
        "ipv4_src": "10.0.25.1",
        "ipv4_dst": "10.0.0.1",
        "eth_type": 2048
    },
    "actions":[
        {
            "type": "SET_FIELD",
            "field": "ipv4_dst",
            "value": "10.0.0.2",
            "eth_type": 2054
        },
        {
            "type":"OUTPUT",
            "port": 1
        }
    ]
}
```

Figure 5.3: Example of command that inserts flow rule

A command can contain the following fields:

- **dpid:** An ID of a switch where the rule is going to be inserted.
- **idle_timeout:** A rule will be deleted after 30 seconds of no traffic matched by this flow.
- **hard_timeout:** A rule will be deleted after 30 seconds regardless of traffic. Duration of timeout depends on the intent we want to achieve.

If both idle and hard timeouts are zero the rule is considered permanent. In our experiments, we inserted only permanent rules.

- **priority:** Sets the priority (integer) of a rule. If a packet matches multiple rules then the rule with the highest priority will be applied.
- **match:** A list of key-value pairs where a key is the name of a field by which a packet will be compared to a specified value. In the example shown in Figure 5.1.1 all packets originating from host with IP *10.0.25.1* and destined to *10.0.0.1* will be matched, and the specified actions will be applied.
- **actions:** Actions that will be applied to packets that match the given rules. In this example a switch with DPID=12 will change the IP destination field from *10.0.0.1* to *10.0.0.2* for all matching packets. Each packet will be forwarded through port 1 of this switch.

Besides inserting rules, we can also delete them in similar fashion with a command depicted in Figure 5.1.1.

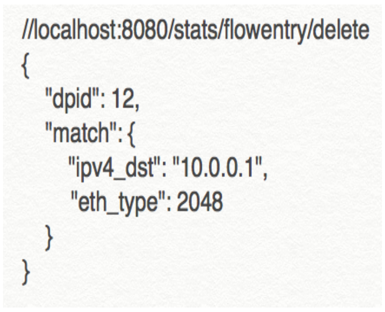

```
//localhost:8080/stats/flowentry/delete
{
   "dpid": 12,
   "match": {
      "ipv4_dst": "10.0.0.1",
      "eth_type": 2048
   }
}
```

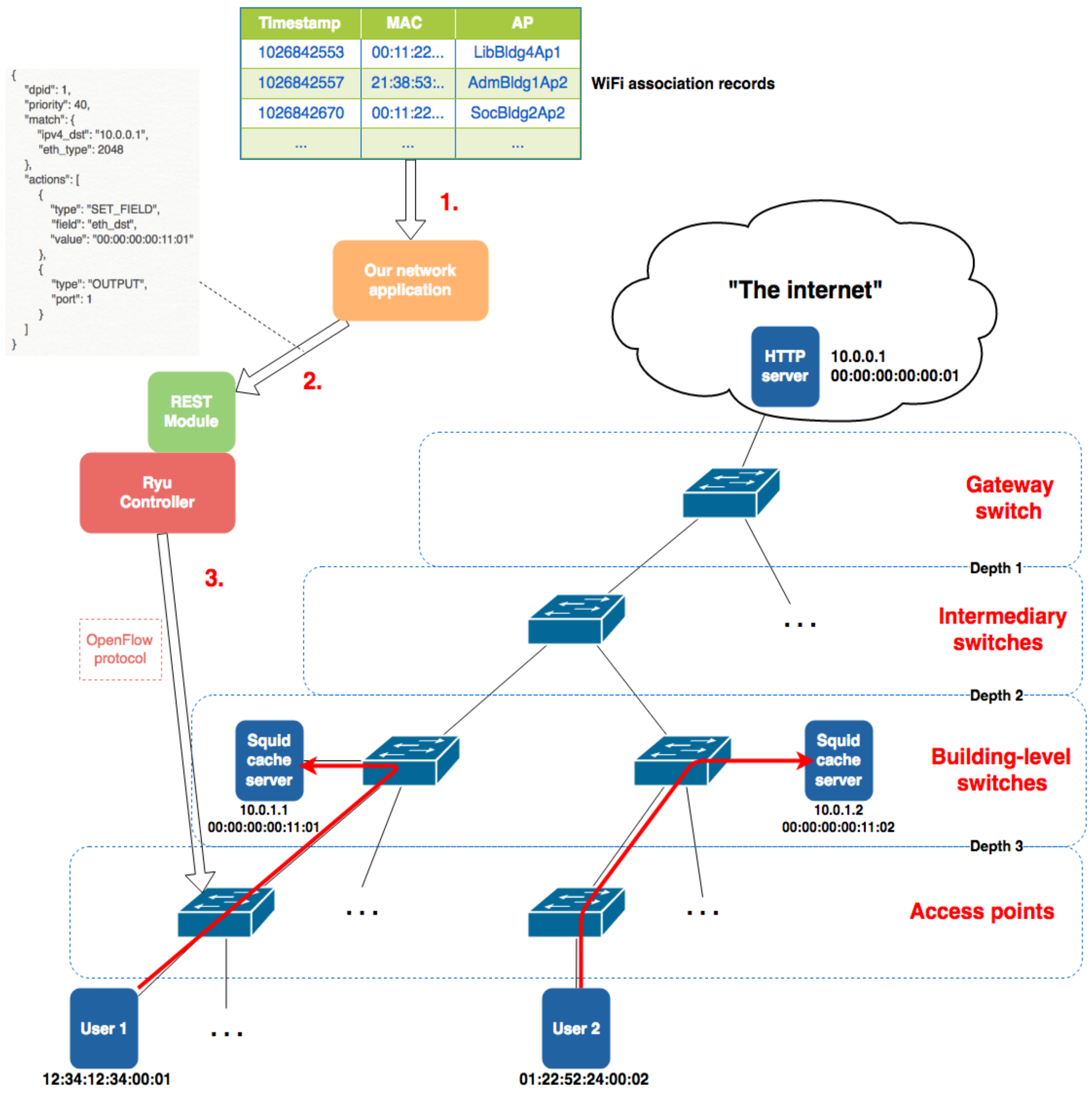
Figure 5.4: Example of a command that deletes a flow rule

The main functionality needed for the implementation of our solution proposed in Chapter 4 is the forwarding of user requests to a cache server located in an arbitrary location in the network. With the Ryu controller there are a few different approaches to redirect a flow to an arbitrary host in the network. First, we could insert flow rules on all switches on the path from an originator to a target host using the appropriate *OUTPUT* action. But an easier and more scalable solution is to change the destination MAC address to a target hosts MAC address. This way switches will automatically forward the packet as if it was intended for the target end-destination from the start. This solution also takes less time to implement since we only need to know MAC addresses of cache servers and DPIDs of access point switches.

5.1.2 SDN simulator tools

In our experiments we use an open-source tool called Mininet to simulate larger SDN networks. Mininet allows us to programmatically create network switches and hosts and connect them into custom topology. We can manage all network nodes via Mininet API or directly via shell. All nodes have unique MAC addresses and are like mini servers so we can run arbitrary code on them. Our first host represents "the internet" and is connected to root switch in our tree topology. On this host we run python's *SimpleHTTPServer*, which serves the content to the users. Users are also represented by Mininet hosts and are connected to switches on the edges of the network (access points). We simulate user activity by making http requests with shell tool called *curl*.
For cache servers we are using open source tool called Squid. It supports HTTP, HTTPS, FTP and other popular protocols. It can be used in regular and reverse proxy mode and it also provides extensive options for hierarchical caching setups. Most importantly, it supports interception caching, which is essential part of our solution since we are using multiple cache servers and changing proxy settings on the client side would simply not be feasible. Squid servers also run on Mininet hosts that we connect to network switches.
We connect SDN switches simulated in Mininet with an instance of the Ryu controller. These connections are not a part of the simulated network topology and controller command packets sent via these links do not interfere with the network data traffic. Controller messages are sent to network switches via the OpenFlow protocol. Finally, the Ruy controller's REST module, de-

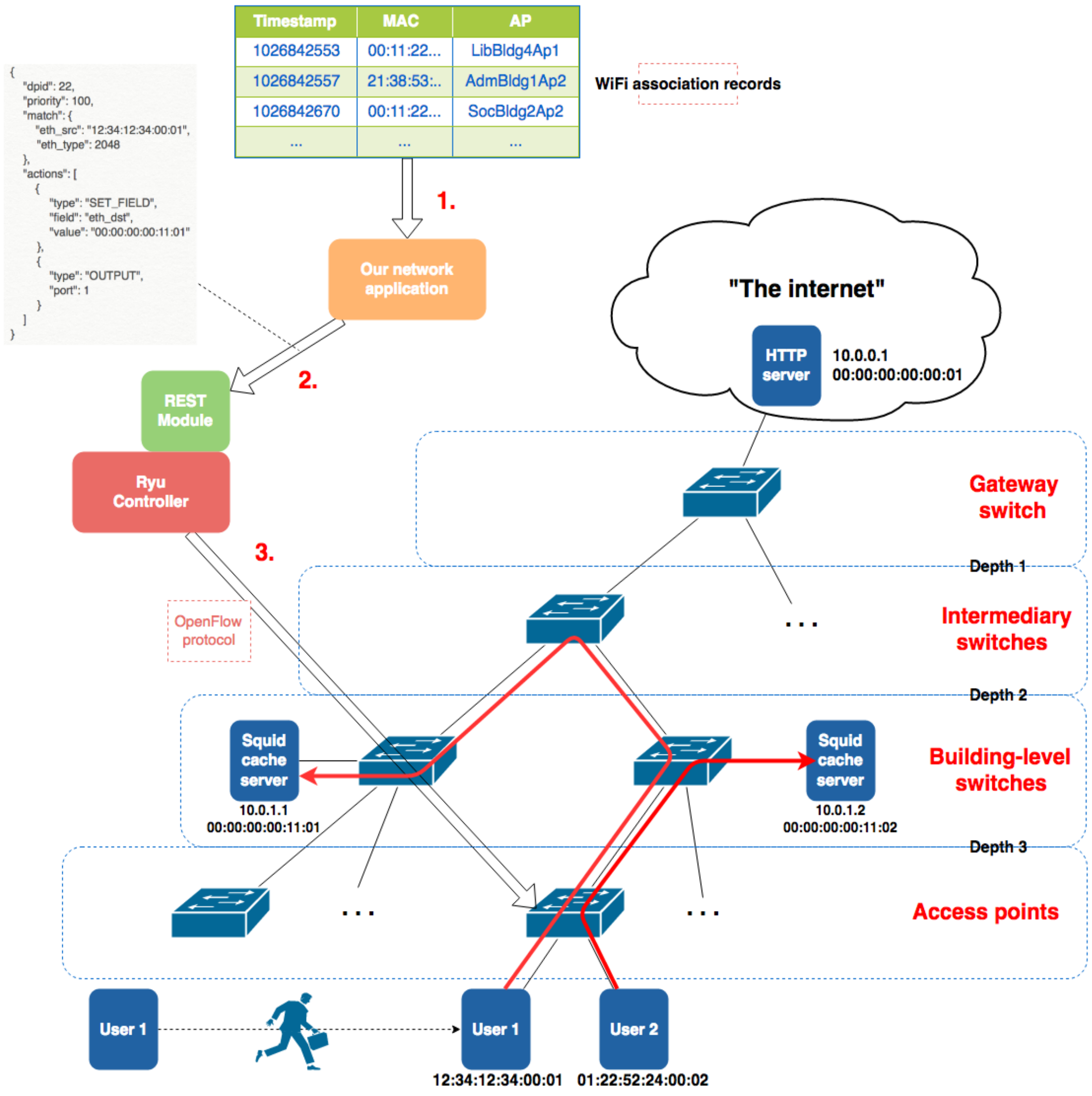
scribed in the previous subsection, parses the JSON commands we issue, and sends specific instructions to appropriate switches.
In the Figures 5.1.2 and 5.1.2 we show the full setup we use in experiments. We show topology for *Building-level caching* experiment, where we have cache servers only on building switches. Our network application receives WiFi association records and computes which cache servers would be optimal for caching the content of specific user. In the example, our algorithm selects the cache server with ip *10.0.1.1* as designated cache server for *User 1* and cache server with ip *10.0.1.2* for *User 2*. Application sends these commands in JSON formats to controller's Rest module where they are parsed and transformed in OpenFlow commands and then sent to specific access point switches where they are inserted in flow table. In the second figure we display how mobility is handled. Our application sends command that will redirect *User 1* to his previous cache server, to second AP switch. Based on this rule, second AP switch will change the destination MAC address of the packets from *User 1* so they will be redirected to cache server with MAC *00:00:00:00:11:01* when *User 1* connects to second AP. This new rule has higher priority than the rule from first figure so this means that only this new rule will be applied if packet matches both rules. The timing of the command is not really important as long as it happens before user makes the transition. We send only permanent commands so the rules stay in the switches' flow tables until we delete them.



Timestamp	MAC	AP
1026842553	00:11:22...	LibBldg4Ap1
1026842557	21:38:53:..	AdmBldg1Ap2
1026842670	00:11:22...	SocBldg2Ap2
...	...	...

```
{
    "dpid": 1,
    "priority": 40,
    "match": {
        "ipv4_dst": "10.0.0.1",
        "eth_type": 2048
    },
    "actions": [
        {
            "type": "SET_FIELD",
            "field": "eth_dst",
            "value": "00:00:00:00:11:01"
        },
        {
            "type": "OUTPUT",
            "port": 1
        }
    ]
}
```

Figure 5.5: Experiment setup (Building level caching).



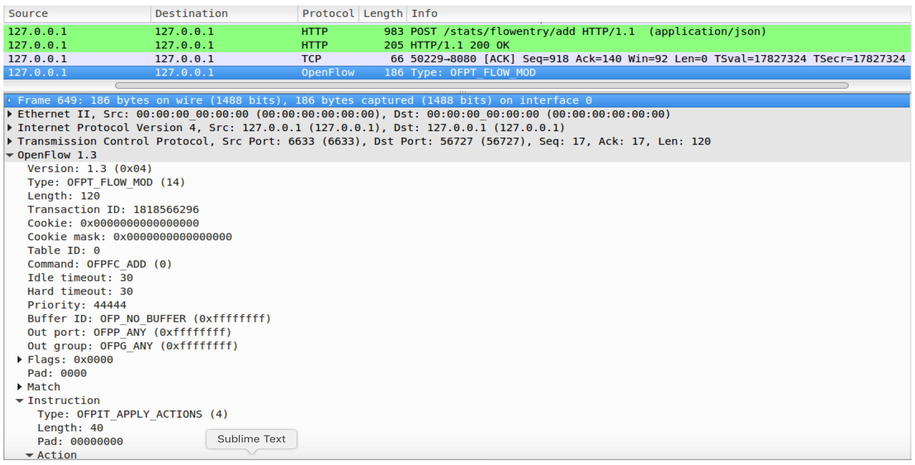
```
{
    "dpid": 22,
    "priority": 100,
    "match": {
       "eth_src": "12:34:12:34:00:01",
       "eth_type": 2048
    },
    "actions": [
       {
          "type": "SET_FIELD",
          "field": "eth_dst",
          "value": "00:00:00:00:11:01"
       },
       {
          "type": "OUTPUT",
          "port": 1
       }
    ]
}
```

Timestamp	MAC	AP
1026842553	00:11:22...	LibBldg4Ap1
1026842557	21:38:53:..	AdmBldg1Ap2
1026842670	00:11:22...	SocBldg2Ap2
...	...	...

Figure 5.6: Experiment setup (Building level caching) - handling user movement.

5.1.3 Measurement tools

The main goal of our work is to reduce the burden that a campus-wide network experiences on its core links, and at the same time, to ensure that user experience remains satisfactory. Thus, the main metrics we measure



```
Source          Destination     Protocol Length Info
127.0.0.1       127.0.0.1       HTTP        983 POST /stats/flowentry/add HTTP/1.1  (application/json)
127.0.0.1       127.0.0.1       HTTP        205 HTTP/1.1 200 OK
127.0.0.1       127.0.0.1       TCP          66 50229→8080 [ACK] Seq=918 Ack=140 Win=92 Len=0 TSval=17827324 TSecr=17827324
127.0.0.1       127.0.0.1       OpenFlow    186 Type: OFPT_FLOW_MOD

▸ Frame 649: 186 bytes on wire (1488 bits), 186 bytes captured (1488 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
▸ Transmission Control Protocol, Src Port: 6633 (6633), Dst Port: 56727 (56727), Seq: 17, Ack: 17, Len: 120
▾ OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_FLOW_MOD (14)
    Length: 120
    Transaction ID: 1818566296
    Cookie: 0x0000000000000000
    Cookie mask: 0x0000000000000000
    Table ID: 0
    Command: OFPFC_ADD (0)
    Idle timeout: 30
    Hard timeout: 30
    Priority: 44444
    Buffer ID: OFP_NO_BUFFER (0xffffffff)
    Out port: OFPP_ANY (0xffffffff)
    Out group: OFPG_ANY (0xffffffff)
  ▸ Flags: 0x0000
    Pad: 0000
  ▸ Match
  ▾ Instruction
      Type: OFPIT_APPLY_ACTIONS (4)
      Length: 40
      Pad: 00000000
    ▾ Action
```

Figure 5.7: Network trace of controller in action

are the delay to serve a user request, and traffic load on different parts of a network. To emulate user requests and measure the service delay we use a data transfer tool *Curl*. It is a command line tool and library and it supports many different protocols like HTTP, FTP, Gopher, SMTP, etc. It also supports file uploads, authentication and proxy tunneling, although we do not use these features in our experiments. For traffic measurement we use *Bandwidth Monitor NG*. We measure the amount of total passed traffic through ports on the switches in our network, although the tool can also count the incoming and outgoing packets and errors. Data can be stored in csv and html format or displayed on the screen. We are interested only in network traffic grouped by link depth so we created simple script which processes total traffic of all ports and groups them by depth. Finally, we use *Wireshark* for debugging. The tool allows us to examine individual packets, including control packets from the Ryu controller to individual switches, as shown in Figure 5.1.3.

5.2 Experimental setup

As a basis for comparison we use simulation results of scenario with one cache server placed at the gateway. We also display the results where no caching was used, to put all the gains and trade-offs into perspective. Our problem domain consists of many different parameters which affect the results of our simulation in various ways. We try to outline the most important ones and provide experiment results to show the effect of each covered parameter.

5.2.1 Network topology

We experiment with two different topologies. First, we show negative impact of user mobility on a small network mimicking the structure used in Figure 3.1 where we devised our algorithms. Second, we use a large topology that represents a realistic picture of an actual campus network setup. The topology is based on a well known Dartmouth Campus dataset [17]. This dataset includes system logs, Simple Network Management Polling data, tcpdump data and records of access point association of each wireless card seen on campus. Association records, which we use in our experiments, were collected between 2001 and 2003 and contain data for over 600 APs and several thousand users at Dartmouth College. We choose this dataset because it contains a list of association records from which we can construct movement traces and also a network topology from coordinates of access points. Consequently, on the basis of this information we can craft a realistic simulation of mobile Internet users moving through a real campus network. The main challenge, however, is to overlay the campus network topology on top of the information we have – AP location only. We took an approach based on hierarchical clustering of APs fixed at their latitude/longitude coordinates. First, since some APs are missing the location information, we interpolated this information from other APs within the same building. Then, we grouped APs by buildings and calculated approximate building center coordinates by averaging the coordinates of APs. We put a switch at each of these centers, and in Figure 5.8 we show their geographical distribution. Points represent computed location of buildings with WiFi access points on the campus in 2003.

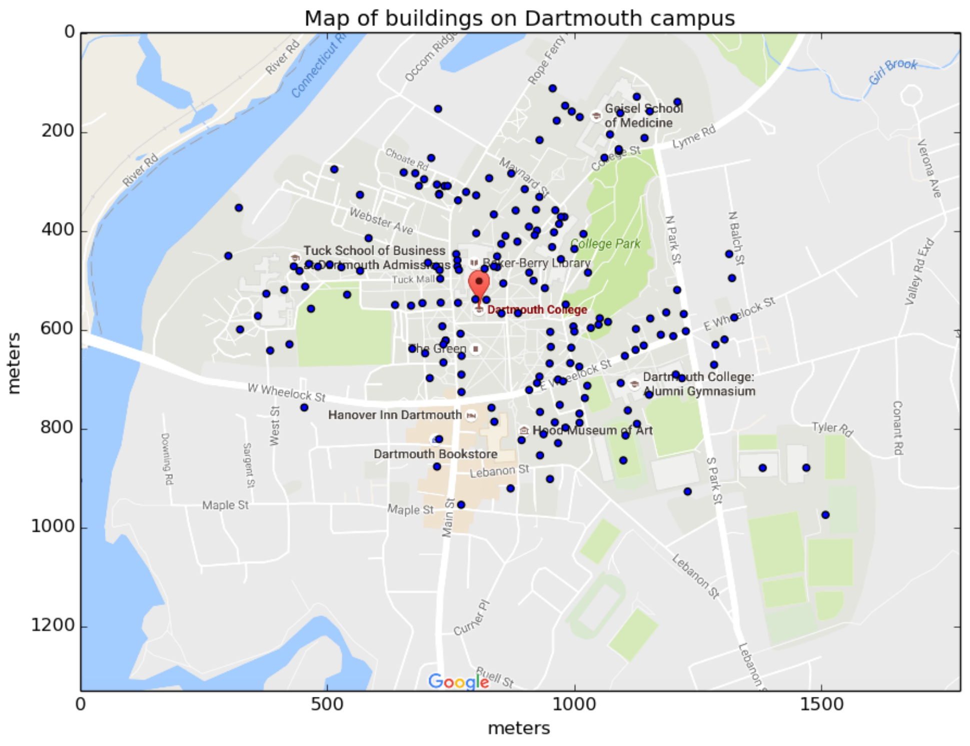


Figure 5.8: Buildings on Dartmouth campus

We then generated the switch linkage matrix with the “ward” hierarchical clustering method [25] using the Euclidean distance metric. This matrix contains links between clusters, hierarchically arranged, thus, can also be visualized with a dendrogram, i.e. Figure 5.9. Our reasoning is that buildings that are close together are also connected to the same parent switch. Validating whether our approach results in a network that indeed mimics the actual Dartmouth campus WiFi network was not possible, as we have no information on the core network connections. However, our goal here is to produce a reasonably realistic estimate, and we believe that the above approach is appropriate.

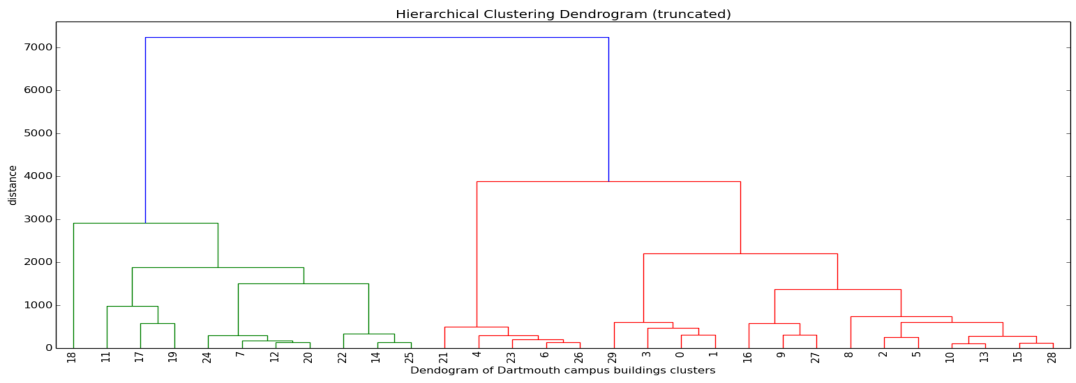


Figure 5.9: Dendrogram of generated building clusters

The Dartmouth Campus trace is rather extensive with 623 APs. Due to the limitations of the experimental setup (in particular Mininet), we concentrate on a topology built on top of 100 most heavily used APs. From the resulting dendrogram we obtained 170 switches in total: 30 intermediary switches, 40 switches representing buildings, 100 switches representing access points. Each switch is connected to it's own host, which represents a potential cache server. On the root switch we have an additional host proxying for the public Internet and running an HTTP server – Python *SimpleHTTPServer*. To better simulate real world environment we also set a constant *50ms* delay on the link between root switch and the "Internet" host. Users are represented by hosts connected to AP and their movement is simulated by disconnecting the host from one AP and connecting it to the other AP. Finally, WiFi connection, by the nature of wireless, has higher packet loss than wired core network connection. This plays an important role in our experiments since retransmissions can take different times depending on whether they need to complete a path to an intermediary cache server or the gateway. To simulate wireless losses we enforce 1% packet loss on all links between APs and end-users.

5.2.2 Caching

In this thesis we focus on improving utilization of **forward proxy caches** in a private network. To make our experiment as representable of real world scenario as possible we need to pick appropriate cache parameters. We are using open source tool Squid cache proxy `http://www.squid-cache.org`. Although such proxies have a lot of configurable parameters, we will focus only on few that we think are the most important:

- **Cache size**. Amount of disk space that can be used to store the content. This is important parameter because the more content we can store, more bandwidth we can save and lower overall delay experienced by user. We must be extra careful in our simulation because our content pool is limited and we should pick cache size as a percentage of our content pool. We do not want to have all our content cached as this would produce results that have no basis in reality. In all our experiments we select such cache size pool that average cache hit rate is between 20% and 35% which we believe is possible scenario in various real world environments.

- **Cache replacement policy**. The cache replacement policy parameter determines which objects are evicted (replaced) when disk space is needed. Squid offers four algorithms [23]: **LRU** (Least recently used), **LFU** (Least Frequently used), **heap GDSF** (Greedy-Dual-Size-Frequency) and **heap LFUDA** (Least Frequently Used with Dynamic Aging). Although the subject of replacement policies is not the goal of this thesis, we do run an experiment where we compare the different replacement policies and pick the one that performed best in our scenarios.

- **Memory replacement policy** determines which objects will be removed from memory when space is needed. Same policies are supported as above.

- **Cache server placement**. This is not a cache parameter but a strategic decision when designing a network topology. This is a mathematical problem and is a member of facility location problems (k-median) [1]. We would like to point out that this is not the original problem we are trying to solve despite the problem being in the same problem group. The results of our algorithm could still provide some insights on the proxy placement problem. But core of our solution is Reverse Greedy algorithm which does not necessarily return an optimal solution. Since this is a design problem which would have long standing effect on the network, we would argue that different, more precise algorithm using Primal-Dual schema[13] or Linear Programming Rounding approach [4] would be more appropriate in this case.
  Most simple example would be having a cache server on the gateway switch. Problem with this approach is that core network links are still fully burdened and there is higher probability of congestion if demand increases beyond capacity of any link in the network path. In other words, we decrease the load only on the backhaul link. On the other side we could place cache servers on all of APs. We find two main problems with this approach. First problem is that there would be a lot of duplicate content over all the caches, because users on different APs may request the same content [26]. We experiment with following placement schemes. **Caching on all switches** - we place cache server on all switches. Of course this is not the most realistic setup, but it provides some insights about the performance of our algorithm. We compare this to the static **Caching on the gateway** placement scheme. In separate experiment we test a compromise between gateway and edge caching - **Building-level caching**. We place cache servers only on building switches (Figure 5.1.2) and then we compare our solution with static caching scheme (no redirections for mobile users).

Forward proxy has two modes of operation:

- **Regular mode**. In this mode user must change the settings in the browser. Browser then talk with the proxy. User is aware that proxy is acting as a middleman. This is more clean solution, but the drawback is that it needs manual settings on the client side. This makes it unsuitable for our case.

- **Interception or transparent proxying**. In this mode network is configured to forward the request to proxy which then "intercepts" the request. This time user does not know that packets are being intercepted. This solution does not need any changes on clients, but it has other disadvantages explained in the Squid manual [11] like **Breaking TCP/IP standards** because user agents think they are talking directly to the origin server, **breaking IP authentication** because packets received by the origin have source IP of Cache's IP address, **doubling DNS load** because client and proxy have to do their own DNS lookup, etc.

5.2.3 Movement patterns

The Dartmouth Campus dataset contains records showing the AP association of each wireless card seen on campus [17] between 2001 and 2003. Each file in the dataset represents one wireless card and it contains record with timestamp and the AP name. If record contains *Off* this means that wireless card was disassociated.

Example: File *00a0f87c2ba7.log.mv* contains

```
1026842553 LibBldg2AP7
1026842579 AdmBldg16AP1
1026842620 OFF
...
```

We restrict our simulation to data collected during Fall 2003 period as it contains enough records to provide a thorough evaluation of our solution.

First, we parse files of all wireless cards and join all association records into one file with the same format as above, but with and additional user ID field. We filter the records to the previously selected APs, and further selected the top 100 most active users per each of the access points. The end result is a list with 100.000 association records.
The above data gives as the actual movement of the users in a network. In our work we are interested in the performance of our proposed caching strategy, and assume that user movement prediction is done by a separate module. Thus, in the beginning we experiment with caching solutions adjusted to perfectly predicted user movement. Yet, we acknowledge that in reality actual movement patterns may deviate from the prediction. Therefore, we also implement a *Discrete-time Markov Chain (DTMC)*, train it on the movement from a portion of the data, and then generate future movement by querying the chain. We construct DTMC by going through of the first ninety percent of the rows in our dataset and counting transitions between states in a matrices of size $|APs| \times |APs|$. In final step we divide by each counter by the total number of transitions of respective AP. We then use this DTMC to generate the movement that need not correspond to the actual movement (of the rest of the trace). It is worth mentioning that our dataset was collected in 2003 and the majority of the records are from laptop wireless cards. This means that data does not necessarily represent movement patterns that would be observed in a campus network today. For example, smartphone usage is heavily underrepresented in the Dartmouth trace. To make the generated trace more representative, and more challenging for our algorithms, we increase the mobility by lowering the probability of self repeating states and evenly increasing the non-zero probability of the transitions.

5.2.4 Request patterns

The Dartmouth dataset unfortunately does not contain any information about the Web content that was requested. We also could not find any other dataset that would contain such information. Instead we rely on a study that investigates the distribution of accessed videos on Youtube [21]. The distribution the authors have identified and used in their experiment is shown in Table 5.2.4. We use a scaled down version of the distribution to shorten simulation time, and show its histogram in Figure 5.11.

Percentage	Original size	Scaled down size
40%	5 MB	10 kB
30%	10 MB	20 kB
20%	15 MB	30 kB
10%	25 MB	50 kB

Figure 5.10: Content size distribution

To setup a realistic content access distribution across users we refer to a study by Wolman et al. performed in campus environment of the University of Washington, USA [26]. The study focuses on documents sharing patterns in an organization and demonstrates a strong temporal locality of multimedia downloads. Their findings show that about 75% of requests have already appeared at one point in the trace. This is confirmed by another unrelated study [10]. Furthermore, the study finds that **40% of requests to the shared documents we repeated by the same user** [26]. This interesting characteristic gave us the idea that we could redirect mobile users to same cache servers they were using on previous locations. Since SDN enables easy flow modifications, this also means that we could push caches servers closer to the edge and still serve mobile users according to this characteristic. To conclude, we modeled request patterns for our experiments based on all the characteristics explained above. Our content pool consist of 4000 cachable pictures 5.11. According to findings in [26], 40% of requested content was uncachable, therefore we added 1400 uncachable files to our content pool.

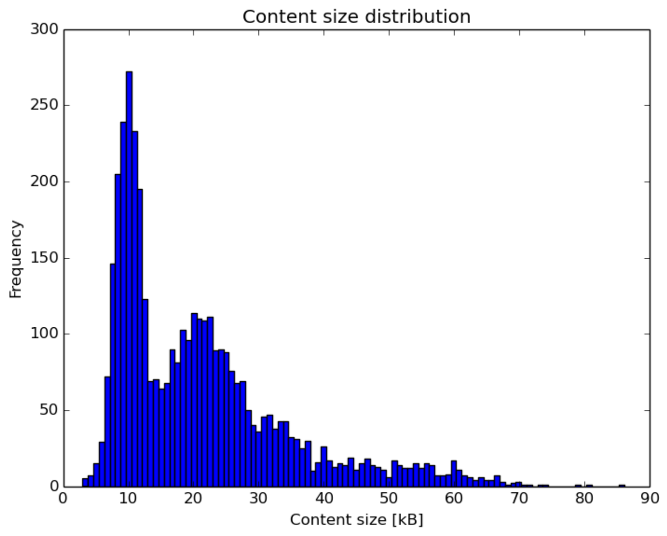


Figure 5.11: Histogram of content size distribution

Chapter 6

Results

6.1 Mobility and hit rates

The field of network caching was extensively researched, but not many experiments take into account the mobility factor of the users, because this aspect became noticeable only in the last few years, with the increasing performance and popularity of smartphones. This mobility factor comes into play when user changes location in a network with multiple cache servers.

To show the impact of mobility we devised small experiment with parameters that we think represent normal functioning cache server in a network. We set up a small topology with two cache servers on two access points, each cache server having 10 MB disk space. On each access point there are 10 users each requesting 4 MB of content of which 20% of requested content is shared between users. Request patterns of users are modeled based on the findings in the Wolman's thesis [26] that 40% of requests to the cached content come from the same user. We are interested in the characteristic of the cache hit rates when a user moves between cache servers, so we will measure hit rate experienced by single user that was first using one cache server for a longer period of time and then moved to another cache server used by another group of users. We do this to avoid initial cold start problem, where the cache is empty and hit rates characteristics are different.

In the first part of the experiment users are static and are evenly requesting files from their designated content pool until both cache servers fill up and we reach intended average 30% hit rate, which can be seen in the first part until the vertical line in Figure 6.1. Then we moved single user to the other AP and we measured and plotted the moving average of the cache hit rate for this user. In Figure 6.1 we can see the hit rate drop after user was moved and the increase back to 30%. When a user is static and is using one cache server for longer period of time, the cache fills with a portion of his content pool resulting in around 30% hit rate. When a user moves away to another AP under provision of different cache server, the hit rate first drops to around 10% and then starts slowly rising back to 30%, while new server starts caching the objects from user's content pool. The reason for 20% drop is the fact that new cache server holds objects from other users' content pools and only small percentage is shared with the new user. Of course the actual values like hit rate threshold, slope of the drop and the rise depend on the network environment (cache size, number of users on each cache servers, request patterns) but the characteristic stays the same under "normal" operating conditions. Of course this negative impact applies only to scenarios with multiple cache servers.

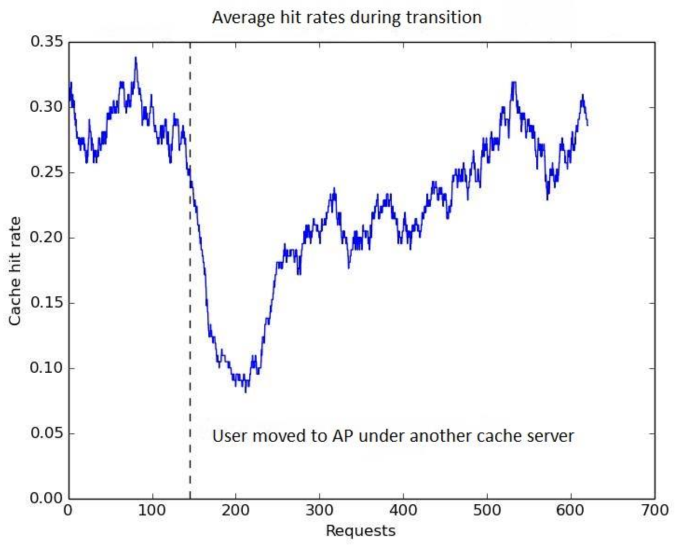


Figure 6.1: Cache hit rates during transition - user perspective

To reduce the traffic on the core network links we try to select cache server closer to the edge so requests do not need to travel all the way to the gateway. For mobile users this introduces the cache hit rate penalty seen in Figure 6.1. In order to “support” mobility we must minimize the effect of this hit rate drop. Our proposed solution does this by analysing user’s movement traces and designating few best placed cache servers to this user. In practice this means that cache servers near most frequently used APs are chosen for this user and when the user moves to another AP, his requests are redirected to the closest designated cache server, avoiding the hit rate drop but taking a longer route.

6.2 Individual node level caching

In the next experiment we compared performance of our solution with the performance of the most simple caching scheme, one cache server placed on the gateway. In order to get fair comparison, we lowered the disk space of cache servers in our solutions, because we are using more of them. Ex-

Parameter	Value
Topology	170 switches (100 access points)
Content pool size	4000 cachable images (total of 146 MB)
	1200 non-cachable images (total of 58 MB)
Cache pool size	20 MB
Cache replacement policy	LRU
Number of users	100
User content pool	400 images per user
Movement pattern	fixed

Table 6.1: Parameters used in experiments in this chapter

periment was done on larger topology, generated from Dartmouth dataset. Topology generation is described in subsection 5.2.1. We used parameters described in table 6.1. To quantify the benefits our solution provides for the network, we measure network traffic on all network links . Improvements for user experience are quantified by measuring delay of requests.

Note: *Cache pool size* consists of disk space and memory. Through our exploratory testing we concluded, that although memory caching is naturally much faster than caching on the Solid-state drive, the difference did not have any noticeable impact on the results. We assume the impact would be noticeable if we used Hard-disk drive. We decided to allocate 90% of *Cache pool size* to the disk and 10% to the memory, because it disk capacities are generally much larger that RAM capacities in general-purpose computers.

6.2.1 Network traffic

First results show noticeable traffic reduction of our solution comparing to basic caching on the gateway method (Figure 6.3), although we have to emphasize that movement pattern is known in advance, so these results are from “best case” scenario. Single median solution, explained in section 4.1, showed the least improvement. This was expected, since we still use only one cache server. Cache hit rate is the same as with gateway caching example, because users still connect to one cache server, only the location of this cache server is optimized for user's requests. With the location change we lowered the in-network traffic between 2% and 10%, but backhaul link received the same amount of traffic as gateway cache server, because hit rates did not change. More interesting are the results for scenarios with multiple cache servers per user. Surprisingly, the scenario with two cache servers (2-median) per user, results in lowest traffic on all depths, including the backhaul link. To explain this, we analysed movement patterns and we discovered that an average user was mostly staying at two different locations. Because adding more cache servers lowers the individual cache size, additional cache servers actually lower the overall bandwidth savings since only two of them are used the most. Using more than three cache servers per user has given negative results. Although in-network traffic was reduced due to cache servers being selected closer to the edge, backhaul link received more traffic, because smaller cache size resulted in lower hit rates comparing to other methods (Figure 6.2).

Scenario	Cache hit rate	Backhaul bandwidth savings
Root cache	32.3%	130.2 MB
Single median	32.3%	130.2 MB
2-median	37.6%	171.1 MB
3-median	34.4%	165.8 MB
4-median	28.1	100.3 MB

Figure 6.2: Hit rates for experiment with fixed movement pattern.

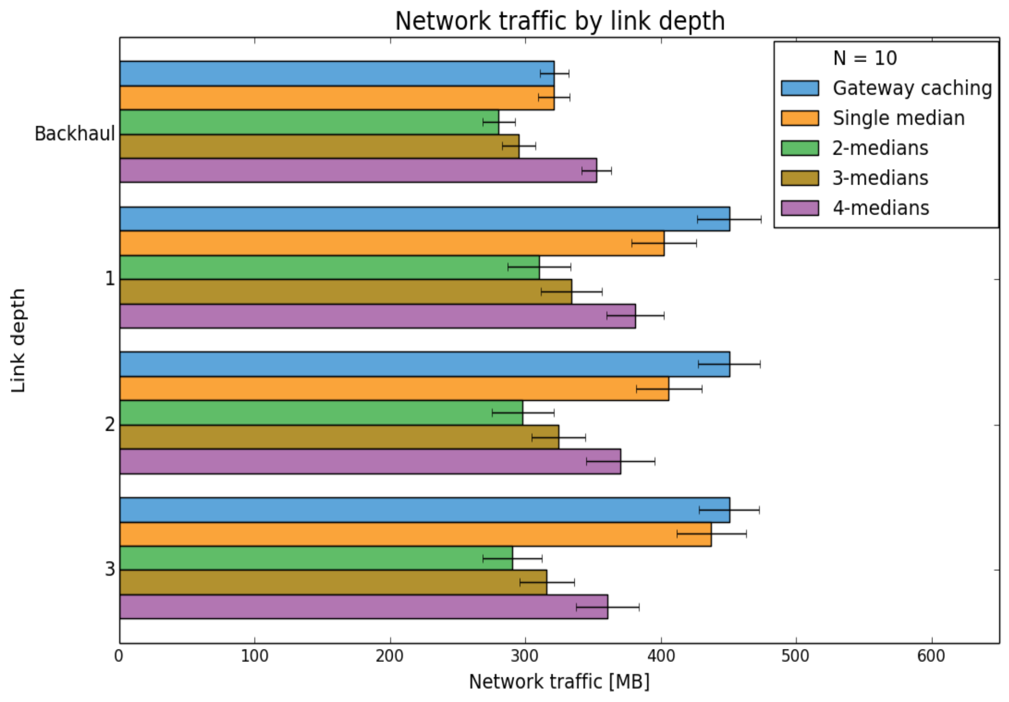


Figure 6.3: Network traffic in experiment with fixed movement patterns.

Second experiment was done with the same parameters as the first (Table 6.1). We only replaced fixed movement patterns with randomized version (subsection 5.2.3). We constructed Markov chain for each user to make their movement pattern probabilistic and make experiment scenario more realistic.

From the results we can see that all our solutions are sensitive to change in movement patterns (figure 6.4). Results are showing around 5% increase in overall network traffic when using Markov chain based movement patterns. Interesting result is high increase of 2-median solution, which performed the best in first experiment. As we mentioned before, users in our movement trace seem to mostly stay at two or three APs for a longer period of time. Because of this characteristic, solutions with two and three cache servers select cache servers near most frequently used APs. With probabilistic movement patterns actual request numbers on APs change and now some of the requests need to travel longer path to selected cache servers. This is the same reason that 4-median scenario increases the traffic the least. Four cache server are selected for each user and there is higher chance that an AP that gained some "requests" will be near one of these cache servers.

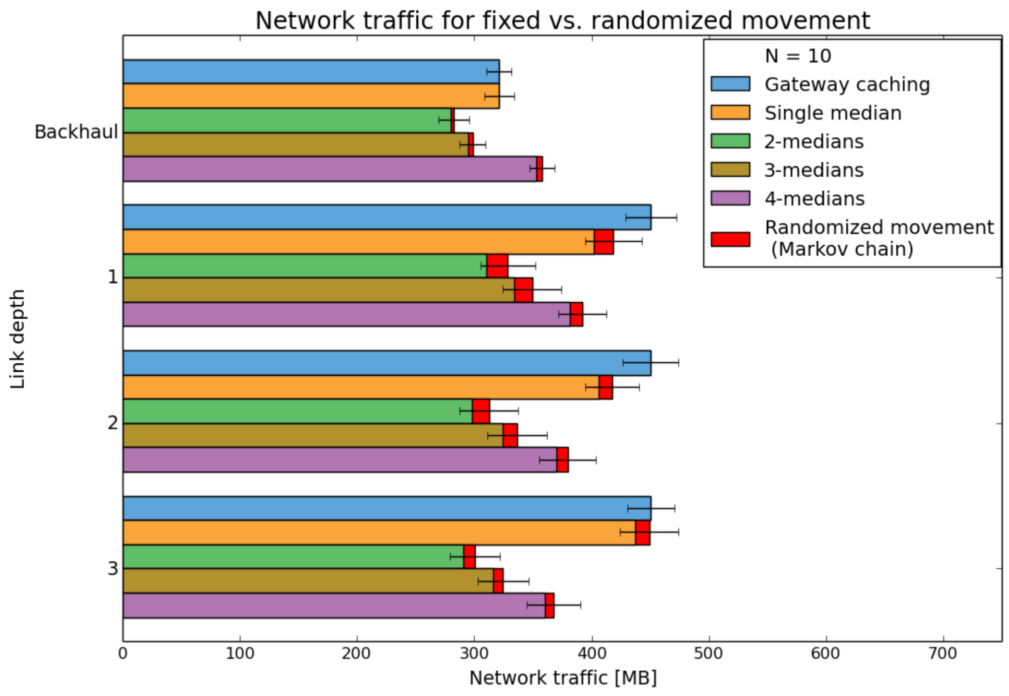


Figure 6.4: Network traffic in experiment with randomized movement patterns (Markov chain)

6.2.2 User experience

As user experience metric we measured the duration of requests with command *curl -w '%time_total'* ... From gathered delays we plotted cumulative distribution functions of delays comparing individual solutions. We also experiment with the effect of cache replacement policies showing the results for each cache policy with other parameters unchanged.

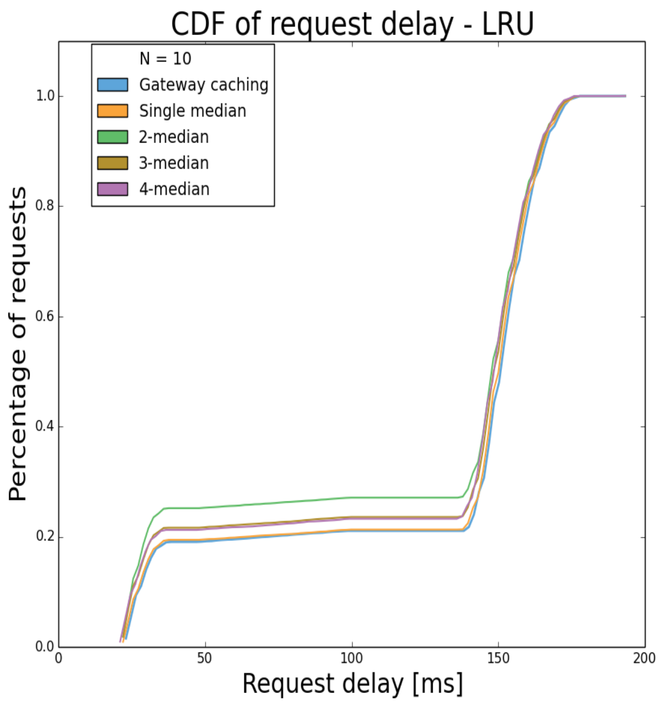


Figure 6.5: Cumulative distribution function for Least Recently Used cache replacement policy.

First figure 6.5 complements the results from previous subsection where we analysed network traffic. Here we can see the difference between solutions with one and multiple cache servers. Largest benefits for the user produced 2-median solution, but we must again emphasize that optimal number of selected cache servers depends mostly on movement patterns of the users. We can see similar results with other cache replacement policies (Figures 6.6 and 6.7), except that absolute values are different.

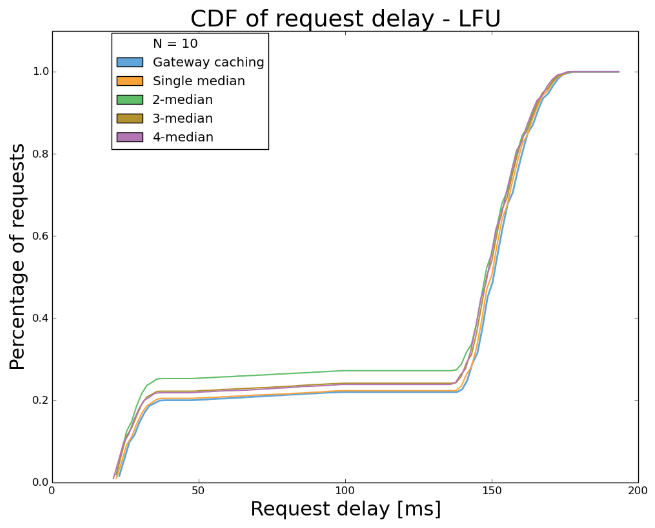


Figure 6.6: Cumulative distribution function for Least Frequently Used cache replacement policy.

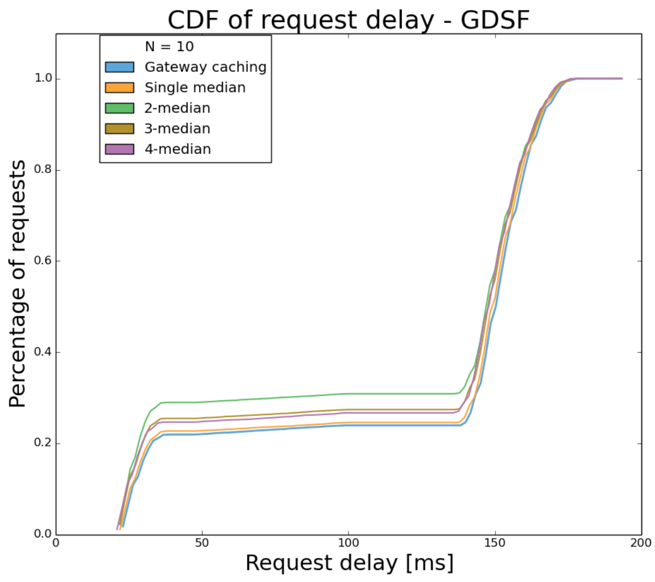


Figure 6.7: Cumulative distribution function for Dual-Greedy-Size-Frequency cache replacement policy.

6.2.3 Choosing the best cache replacement policy

In previous subsection we displayed experiment results for individual cache replacement policy used and we observed that chosen replacement policy did not drastically changed the outcome of the experiment, however if we directly compare replacement policies we can see some interesting findings. We plotted the best performing solution (2-median) and the most basic solution (cache server on the gateway) with different cache replacement policies (fig 6.8). We can see that Greedy-Dual-Size-Frequency policy performed noticeably better than the other two policies. Also interesting is the fact that the difference between *GDSF* and other two policies is higher when 2-median approach was used. Unlike *LRU* and *LFU*, *GDSF* takes into account file size, file access frequency and recentness of last access. Because it operates on more parameters, we assume it can make better eviction decisions, resulting in increased hit rates and byte hit rates (ratio of bytes served by the cache over the bytes requested by the clients) [5].

Based on these results we decided to use *GDSF* replacement policy in the next experiment, where we placed cache servers only on switches representing buildings.

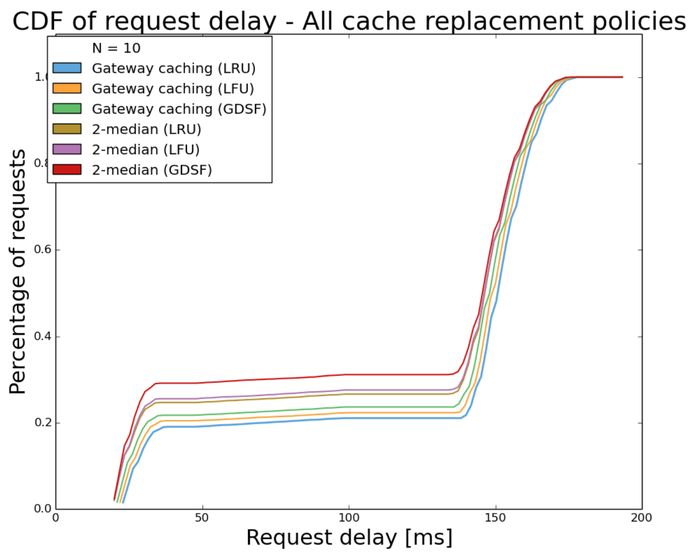


Figure 6.8: Cumulative distribution functions for all cache replacement schemes

6.3 Building level caching

In this experiment we wanted to test our solution in more limited setup. In previous experiments we assumed that cache servers could be placed on all network nodes, which is probably unrealistic assumption. That is why we chose to limit cache servers only on building level switches. Our network application can now choose to cache only on these “building switches”. But now also our basic caching scheme is different. Previously we cached the content on the gateway, but now we have multiple cache servers and this means that mobility of users will cause hit rate penalty. We feel that this comparison is more fair than previous experiment since we do less assumptions of the network and comparison is done on equal parameters.

6.3.1 Network traffic

Results in figure 6.9 show decent traffic decrease for our solutions compared to static caching. We can see that unlike in previous experiments 3-median solution performed slightly better than 2-median. This difference is cause by the fact that majority of users have between two and three "popular" access points where they make most of the requests. In this scenario hit rate penalty is higher if the users with three APs have their requests forwarded to two cache servers instead of three, because cache size are equal on all cache servers. This characteristic does not apply to solutions with four or more cache servers, which means that there are not many users with four popular APs.

Increasing the number of designated cache servers per user decreases the performance, because higher the number of servers more times the user will suffer the hit rate penalty because requests will be served from cache server containing less content of this user.

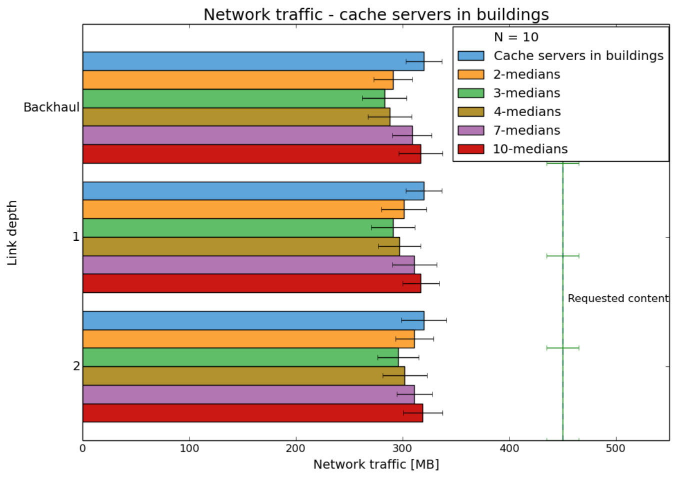


Figure 6.9: Cumulative distribution function for Dual-Greedy-Size-Frequency cache replacement policy.

6.3.2 User experience

In this experiment we also measured request delay and plotted cumulative distribution function for all approaches (Figure 6.10). Results confirm the findings from previous subsection. 3-medians performed best with highest cache hit rates resulting in lower overall experienced latency. We can also see that increasing the number of used cache servers reduces the latency gains by actually making our solution more like static caching.

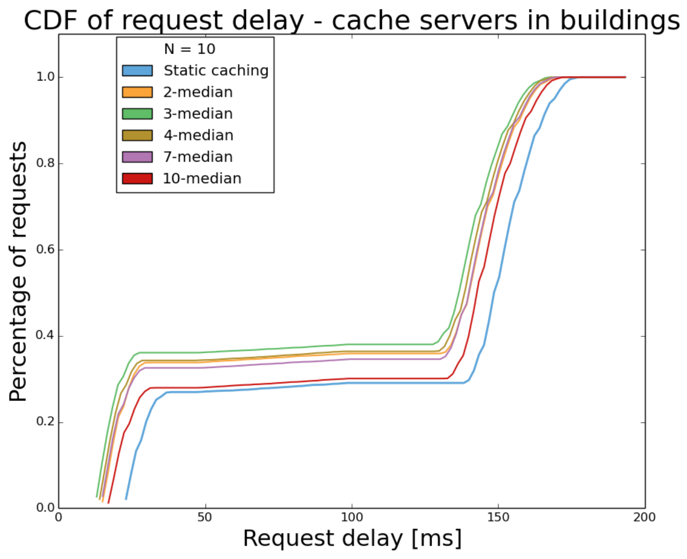


Figure 6.10: Cumulative distribution function for Dual-Greedy-Size-Frequency cache replacement policy.

6.4 Experiment under uncertainty

In previous experiments we assumed that a user's movement does not diverge to much from data traces. In reality user can often move unpredictably. Because our algorithm uses movement patterns for selecting appropriate cache servers, we want to analyse performance under various levels of uncertainty. We do this by artificially adding randomness to test set of our movement pattern. Transition probabilities p_{xn} converge to equal probabilities in each iteration: $p_{xn} \rightarrow \frac{1}{n}$, where the sum of transition probabilities for each access point x is $P_x = \sum_{i=1}^{n} p_{xn} = 1$.

In other words, we lower the higher transition probabilities and increase lower transition probabilities, all converging to same value. In other words, user can move to any AP with equal probability. Results show interesting relation between number of cache servers and randomness of movement, but even more insightful is the implicit connection between number of cache servers and number of most visited APs for a user. As we mentioned before, average user made the majority of requests from two distinctive APs. We could say, that this user has two clusters. It turns out that our algorithm performs best when our k matches the number of users cluster, as can be seen in figure 6.4 in section *All nodes*. When we start introducing randomness in movement pattern, user's clusters start to dissolve. Most of the user's requests from are then redirected to cache servers that are further away than root cache thus increasing overall network traffic.

In the figure we can notice that higher the number of cache servers, higher the level of randomness that caching scheme tolerates before becoming less efficient than Root cache server. For our top scoring $k = 2$ parameter in deterministic movement scenario 6.4 we can observe that it is actually the least tolerant for deviations in movement patterns, reaching and surpassing gateway caching scheme traffic at 28%.

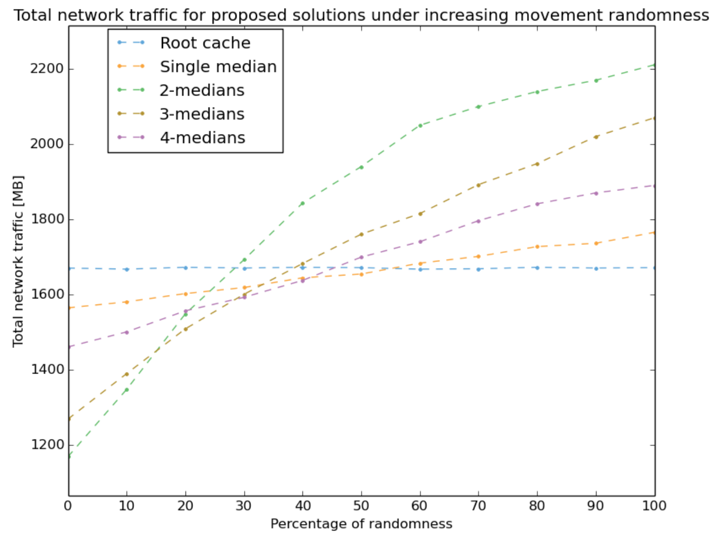


Figure 6.11: Total network traffic for increasing randomness in movement patterns

Another interesting observation can be made about *Single median* approach. The results seem to be in contradiction with our previously discovered relation between k and total network traffic. The reason for this is the fact that Single median algorithm in majority of cases picked cache server near the gateway.

Chapter 7

Conclusion

In this thesis we study the current state of caching in larger WiFi networks and propose a novel solution based on Software-defined Networking, which provides us a platform on which we can easily build more advance solutions. Our primary focus is reducing the network traffic and improving user experience by increasing cache hit rates and thus reducing experienced request delay. More specifically we address the impact of user mobility on traditional caching schemes and provide a dynamic solution that minimizes this negative impact. In the Mobility and hit rates section we show that user experiences a significant drop in cache hit rate when moving to another access point under the coverage of different cache server.

Our network application tries to select optimal cache servers for each user by analysing their historic movement patterns. More requests that user made on specific access point, more likely it is that cache server will be chosen near this access point. When user moves to access point on which he will make only few requests, his request will be redirected to closest designated cache server, which probably already contains a portion of his content pool. Our solution tries to maximize the reduction of network traffic achieved by selecting cache servers closer to the most used access points by a user and minimizing the traffic increase caused by redirected requests from less used access points taking longer path through the network.

We evaluated our solution by comparing it to simple static caching schemes commonly used in traditional networks. In our experiments we used real world WiFi connectivity traces collected at Dartmouth campus [17] which we used to build our network topology and extract movement patterns. In our simulations we measure network traffic on all links, cache hit rates and request delays. In the first round of simulations we compare static caching on the network gateway with our solution that can choose any node in a network as a cache server. Our solution show high traffic reduction, up to 28% in best case scenario using deterministic movement patterns and up to 24% when using probabilistic movement patterns generated with Markov chain method. Results also show that the optimal number of caches servers per user depends mostly on the movement pattern of a user.

User movements are often unpredictable and since our solution relies on accuracy of movement predictions we conducted an experiment in which we artificially added randomness to the user actual movement patterns. We wanted to check the performance of our solution under high uncertainty. Simulation revealed steady decrease in performance with rising level of randomness introduced and eventually resulting in much higher network traffic and higher delay than static caching scheme, which is completely unaffected by the impact of user mobility. Although this experiment is very artificial, we discovered that solutions using higher number of cache servers were less affected by inaccurate predictions than solution using less cache servers.

In the previous experiments we assumed that cache servers are placed on all switches in the network which is valid scenario for research but very uncommon in real world. To test how would our solution perform in more realistic environment we placed cache server only on switches that represent buildings. This means that cache servers are placed very close to the edge of the network. This setup also provides fairer comparison between static caching schemes and our dynamic approach and also better showcases the negative impact of user mobility. Results showed that our solution, tested with varying number of used cache servers, performed up to 9% better in terms of lower network traffic on the backhaul link that the static caching scheme.

In this work we studied the often neglected impact of user mobility on network caching and also proposed a dynamic approach leveraging SDN technologies to minimize this impact, reduce network traffic and improve user experience by reducing request delays. We have showed proof of concept with few experiments that show noticeable gains comparing to static caching schemes. But the field of network caching is very broad, and we see a lot of possible improvements and future research directions. In network caching domain there are a lot of configurable parameters and many different environments. We specifically focused on college campus network because of high predictability of movement and huge dataset available [17]. Main drawback concerning dataset is the age (gathered in 2003) and the fact that it contains only traces from laptop users. Since we study the effect of mobility we think that our solution should be tested with some newer dataset containing appropriate share of mobile smartphone users. We strongly believe, that our proposed solution would perform even better, since mobility impact would probably be bigger. There are also countless combination of various network parameters like topology, cache sizes, replacement schemes, request patterns and others on which we could test our solution.

Biggest improvement of our algorithm would be the implementation of one of the techniques like *Elbow method* [8] or *X-means clustering* [22] for selecting optimal number of cache servers for each user. In the conducted experiments we set the number of cache servers to be equal for all users, but the movement patterns were quite different between them.

Bibliography

[1] Robert Benkoczi, Binay Bhattacharya, Marek Chrobak, Lawrence L. Larmore, and Wojciech Rytter. Faster algorithms for k-medians in trees. In *Mathematical Foundations of Computer Science 2003: 28th International Symposium, MFCS 2003, Bratislava, Slovakia, August 25-29, 2003. Proceedings*, pages 218–227. Springer, 2003.

[2] Matthew Broadbent, Daniel King, Sean Baildon, and Nektarios Georgalas. Opencache: A software-defined content caching platform. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–5. IEEE, 2015.

[3] Martin Casado, Nate Foster, and Arjun Guha. Abstractions for software-defined networks. *Commun. ACM*, 57(10):86–95, sep 2014.

[4] Moses Charikar and Shi Li. *A Dependent LP-Rounding Approach for the k-Median Problem*, pages 194–205. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[5] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy.* Hewlett-Packard Laboratories, Nov 1998. Report number: HPL-98-69 (R.1).

[6] Marek Chrobak, Claire Kenyon, and Neal Young. The reverse greedy algorithm for the metric k-median problem. In *Computing and Combinatorics: 11th Annual International Conference, COCOON 2005*, pages 654–660. Springer, 2005.

[7] Cisco. Cisco visual networking index: Forecast and methodology 2014-2019.

[8] Christopher L. Shook David J. Ketchen. The application of cluster analysis in strategic management research: An analysis and critique. *Strategic Management Journal*, 17(6):441–458, 1996.

[9] M. Dräxler, J. Blobel, and H. Karl. Anticipatory download scheduling in wireless video streaming with uncertain data rate prediction. In *2015 8th IFIP Wireless and Mobile Networking Conference (WMNC)*, pages 136–143, oct 2015.

[10] Bradley M Duska, David Marwood, and Michael J Feeley. The measured access characteristics of world-wide-web client proxy caches. In *USENIX Symposium on Internet Technologies and Systems*, volume 997, 1997.

[11] Mark Elsen. *Concepts of Interception Caching.* squid-cache.org, 2016. http://wiki.squid-cache.org/SquidFaq/InterceptionProxy.

[12] Open Networking Foundation. Openflow switch specification ver 1.5.1. Technical report, Open Networking Foundation, mar 2015.

[13] K. Jain and V. V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 2–13, 1999.

[14] O. Kariv and S. L. Hakimi. An algorithmic approach to network location problems. ii: The p-medians. *SIAM J. Appl. Math.*, 37(3):539–560, 1979.

[15] Hyojoon Kim and Feamster Nick. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, feb 2013.

[16] Mael Kimmerlin, Jose Costa-Requena, and Jukka Manner. Caching using software-defined networking in lte networks. In *2014 IEEE In-*

ternational Conference on Advanced Networks and Telecommuncations Systems (ANTS), pages 1–6. IEEE, 2014.

[17] David Kotz, Tristan Henderson, Ilya Abyzov, and Jihwang Yeo. Crawdad dataset dartmouth/campus, sep 2009.

[18] Kreutz, Ramos, Veríssimo, and Rothenberg. Software-defined networking: A comprehensive survey. In *Proceedings of the IEEE*, volume 103, pages 14–76. IEEE, 2015.

[19] Kwong Yuen Lai, Zahir Tari, and Peter Bertok. Supporting user mobility through cache relocation. *Mobile Information Systems*, 1(4):275–307, 2005.

[20] Bo Li, M. J. Golin, G. F. Italiano, Xin Deng, and K. Sohraby. On the optimal placement of web proxies in the internet. In *INFOCOM 99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications*, volume 3, pages 1282–1290. IEEE, feb 1999.

[21] Afra J Mashhadi and Pan Hui. Proactive caching for hybrid urban mobile networks. *University College London, Tech. Rep*, 2010.

[22] Dan Pelleg, Andrew W Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *ICML*, volume 1, 2000.

[23] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003.

[24] Vasilios A. Siris, Maria Anagnostopoulou, and Dimitris Dimopoulos. *Improving Mobile Video Streaming with Mobility Prediction and Prefetching in Integrated Cellular-WiFi Networks*, pages 699–704. Springer International Publishing, 2014.

[25] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[26] Alastair Wolman. *Sharing and Caching Characteristics of Internet Content*. PhD thesis, University of Washington, 2002.

[27] Kin-Yeung Wong. Web cache replacement policies: a pragmatic approach. *IEEE Network*, pages 28–34, jan 2006.

[28] F. Zhang, Ramakrishnan Xu, Y. Zhanga, Yates Mukherjee, and Nguyen. Edgebuffer: Caching and prefetching content at the edge in the mobilityfirst future internet architecture. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2015 IEEE 16th International Symposium on a*, pages 1–9. IEEE, 2015.