Pedro Miguel Lima de Jesus Vieira

BsC in Computer Science and Engineering

A Persistent Publish/Subscribe System for Mobile Edge Computing

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Hervé Miguel Cordeiro Paulino, Assistant Professor, NOVA University of Lisbon

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

November, 2018

A Persistent Publish/Subscribe System for Mobile Edge Computing

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] `https://github.com/joaomlourenco/novathesis` [2] `http://www.di.fct.unl.pt`

To my mother.

Acknowledgements

First, I would like to start by thanking my thesis advisor, Hervé Paulino, for his complete availability and the many insightful discussions during the concession and development of the ideas presented in this thesis. Hervé also provided me with the necessary hardware equipment and helped me fix some of the bugs I encountered in the simulator, both aspects being absolutely crucial to performing the dissertation's real-life and simulated evaluation scenarios, respectively. A very special gratitude goes out to João Silva. His hands-on approach, whether by brainstorming solutions, suggesting alterations as well as the immense help on the creation of trace files, was very much appreciated. This is extended to all past and present members of the "Hyrax: Crowd-Sourcing Mobile Devices to Develop Edge Clouds" project [1] for their incredible hard work, which I was able to utilize in my solution.

Thanks to the entire FCT NOVA's Informatics department, its teachers and all the valuable knowledge that was transmitted to me.

To my mother, who has sacrificed a lot to provide me with an higher education, for supporting and believing in me, no matter what, but most importantly for raising and shaping me into the man I am today.

A special thank you to my soul mate, Ana Simão, for her constant presence and patience to endure me in my ups and downs. For supporting me, pushing and motivating me to go further even when I was tired and frustrated. For always cheering me up. Thank you for making this journey all worthwhile.

And finally, last but by no means least, also to all the close friends I made during this five-year adventure who positively contributed, one way or another, to my academic and personal success.

Thank you.

Abstract

In recent times, we have seen an incredible growth of users adopting mobile devices and wearables, and while the hardware capabilities of these devices have greatly increased year after year, mobile communications still remain a bottleneck for most applications. This is partially caused by the companies' cloud infrastructure, which effectively represents a large scale communication hub where all kinds of platforms compete with each other for the servers' processing power and channel throughput. Additionally, wireless technologies used in mobile environments are unreliable, slow and congestion-prone by nature when compared to the wired medium counterpart.

To fix the back-and-forth mobile communication overhead, the "Edge" paradigm has been recently introduced with the aim to bring cloud services closer to the customers, by providing an intermediate layer between the end devices and the actual cloud infrastructure, resulting in faster response times. Publish/Subscribe systems, such as Thyme, have also been proposed and proven effective for data dissemination at edge networks, due to the interactions' loosely coupled nature and scalability. Nonetheless, solely relying on P2P interactions is not feasible in every scenario due to wireless protocols' range limitations.

In this thesis we propose and develop Thyme-Infrastructure, an extension to the Thyme framework, that utilizes available stationary nodes within the edge infrastructure to not only improve the performance of mobile clients within a BSS, by offloading a portion of the requests to be processed by the infrastructure, but also to connect multiple clusters of users within the same venue, with the goal of creating a persistent and global end-to-end storage network. Our experimental results, both in simulated and real-world scenarios, show adequate response times for interactive usage, and low energy consumption, allowing the application to be used in a variety of events without excessive battery drainage. In fact, when compared to the previous version of Thyme, our framework was generally able to improve on all of these metrics. On top of that, we evaluated our system's latencies against a full-fledged cloud solution and verified that our proposal yielded a considerable speedup across the board.

Keywords: Mobile Edge Computing, Infrastructure, Android, Framework, Distributed

Storage, Peer-to-peer, Publish/Subscribe

Resumo

Recentemente, tem-se observado um enorme crescimento de utilizadores de dispositivos móveis e, apesar das suas características do *hardware* sejam cada vez mais otimizadas ano após ano, as comunicações móveis ainda são consideradas como um *bottleneck* para a maioria das aplicações. Isto deve-se parcialmente à própria infraestrutura *cloud*, que representa um ponto central de comunicação em que todas as plataformas competem entre si pelo processamento e taxa de transferência dos servidores. Adicionalmente, as tecnologias sem fios habitualmente utilizadas em ambientes móveis são, por natureza, instáveis, lentas e suscetíveis a congestionamento.

Para atenuar a sobrecarga inerente às comunicações móveis, o paradigma "Edge" foi recentemente introduzido, com o intuito de aproximar os serviços da *cloud* aos consumidores, através da disponibilização de uma camada intermédia entre os dispositivos e a própria infraestrutura *cloud*, resultando em tempos de resposta bastante mais rápidos. Sistemas Publicador/Subscritor, como o Thyme, têm sido recentemente propostos como métodos eficazes para a disseminação de dados em redes "edge", devido à escalabilidade das suas interações. Ainda assim, depender apenas de comunicações *P2P* não é factível em todos os cenários devido às limitações de alcance dos protocolos de redes sem fios.

Nesta dissertação propomos e desenvolvemos o Thyme-Infrastructure, uma extensão ao Thyme, que utiliza nós estacionários que estejam disponíveis na infrastrutura "edge" para melhorar a performance dos clientes móveis dentro de um *BSS*, ao transferir uma porção dos pedidos para serem processados pela infraestrutura, mas também para conectar múltiplos grupos de utilizadores dentro do mesmo complexo, tudo com o objetivo de criar uma rede global de armazenamento persistente ponta-a-ponta. Os nossos resultados experimentais, ambos em cenários simulados e reais, mostram tempos de resposta adequados a utilizações interativas e um baixo consumo energético, permitindo que a aplicação possa ser utilizada em diversos eventos sem que apresente gastos excessivos de bateria. De facto, quando comparada à versão anterior do Thyme, a nossa *framework* apresentou, de forma geral, melhorias em todas estas métricas. Para além disto, avaliámos as latências do nosso sistema face às de uma solução implementada na *cloud* e verificámos que a nossa proposta exibiu um *speedup* considerável em todos os aspetos.

Palavras-chave: *Mobile Edge Computing*, infraestrutura, *Android*, *framework*, armazenamento distribuído, ponto-a-ponto, publicador/subscritor

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context & Motivation

In recent times, we have seen an incredible growth of users adopting mobile devices and wearables, such as smartphones and smartwatches, for communicating with friends, performing day-to-day activities or even work-related procedures, going from 1.46 billion users in 2013 to 2.89 billion customers in 2017 [2] (Fig. 1.1) and this trend is poised to keep rising every year.

In 2016, analysts disclosed that mobile devices finally surpassed desktop in terms of internet usage worldwide (mobile traffic accounted for 51.3% of the global interactions, while desktop traffic represented 48.7%) [3]. Although this is generally not the case in developed countries, where desktop usage is still widely prevalent, third world countries often have the highest smartphone usage percentage, like India for instance, reaching to values up to 79% of the country's internet usage [4].

Consequently, along with the soaring introduction of mobile-tailored applications and services as well as the paradigm shift of working on-the-go, Cisco predicts that the global traffic originated from non-stationary devices will grow 600% to 49 exabytes per month in 2021, from the 7 monthly exabytes value recorded in 2016 [5] (Fig. 1.2). Furthermore, the researchers convey that the exponential mobile traffic growth is directly linked to the expected increase in consumable contents, namely (live) video, that is bound to represent the largest chunk percentage in mobile interactions, as well as the mainstreaming of data-heavy experiences, in the likes of Virtual and Augmented Reality (AR & VR) online applications. In the same research, the company also notes that mobile clients will increasingly start to offload their online interactions and data requests to nearer infrastructure stations, such as public Wi-Fi hotspots, instead of using cellular networks. Based on this findings, they are predicting that the deployment of Wi-Fi hotspots will grow

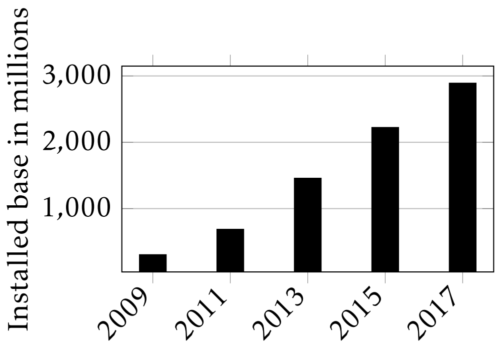


Figure 1.1: Smartphones worldwide installed base from 2009 to 2017. Adapted from [2].

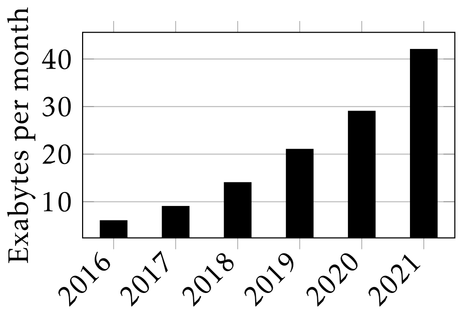


Figure 1.2: Monthly exabytes produced by mobile users. Adapted from [5].

six-fold to 541.6 million in 2021 (up from 94.0 million in 2016) in order to accommodate this offloading trend.

Even though smartphones' hardware capabilities have greatly increased year after year, mobile communications still remain a bottleneck for most applications, both on the client and the server side. Regarding the servers aspect, while using geo-replicated data centers do indeed help in minimizing latency, the company's cloud infrastructure represents a communication hub which potentially receives messages from every kind of client, mobile or desktop, effectively causing the requests from all platforms to compete with each other for the servers' processing power and channel throughput. Additionally, by forcing the requests to go through the entire network infrastructure in order to reach the company servers, it obviously carries additional routing and propagation delays. On the other hand, wireless communication technologies used in mobile environments — such as Wi-Fi, Bluetooth and 3G — are unreliable, slow and congestion-prone by nature when compared to the wired medium counterpart. By adding the characteristics from both sides, mobile interactions are particularly susceptible to the back-and-forth communication overhead, which is particularly crucial to eradicate in delay-sensitive applications such as the ones from the AR and VR categories mentioned previously.

In order to increase performance for mobile applications and interactions, there has been a major paradigm shift, specifically the introduction of "Edge networks". This novel concept brings cloud services to the edge of the network, i.e. closer to the customers, by leveraging the storage and processing capabilities of multiple decentralized "mini-cloud" devices with the purpose of providing an intermediate layer between the end devices and the actual cloud infrastructure.

With the continuous rise of mobile devices, sensors, actuators and the huge amount of generated data as a byproduct of that, Edge nodes are deployed in places like factory floors, vehicles and on top of buildings with the purpose of quickly processing information and act accordingly, if needed. As a consequence of their proximity to the data source and by eliminating the back-and-forth traffic between the devices and the cloud, nodes are able to provide much faster response times since the requests do not need to go through the entire global infrastructure, saving backbone bandwidth and minimizing the occurrences of bottlenecks in the network and the cloud service itself [6, 7].

There are currently three types of Edge layer implementations, with distinct actors, proximity scale and communication protocol specifications, namely Mobile Edge Computing (MEC), Fog Computing (FC) and Cloudlet Computing (CC) [8, 9] (Table 1.1).

Table 1.1: Comparison of Edge layers. Adapted from [9].

	Node Devices	**Proximity**	**Communication Protocols**
FC	Routers, Switches, Access Points, Gateways	One or Multiple Hops	Bluetooth, Wi-Fi, Mobile Networks
MEC	Servers running in base stations	One Hop	Mobile Networks
CC	Data Center in a box	One Hop	Wi-Fi

The Fog Computing paradigm presents a computing layer that mainly utilizes devices like gateways and wireless routers, which are used to compute and store data from end-costumers' devices locally before forwarding to the cloud. MEC, on the other hand, proposes the deployment of intermediate nodes with storage and processing capabilities in the base stations of cellular networks in order to offer cloud computing capabilities inside the Radio Area Network. The Cloudlets are based on dedicated devices with capacities similar to a data center but on a much lower scale present in close proximity to the consumers [9].

Although these Edge computing approaches were developed with a cloud backend connection in mind, other implementations have been presented that do not need any kind of server infrastructure, such as [10]. Furthermore, other iterations are moving away from single node access points to using the client's mobile devices as actual Edge nodes, harnessing their processing, storage and routing/communication capabilities, in order to create a complete local system.

Ultimately, implementing an Edge infrastructure can bring huge improvements to end-users but, due to its novelty, this paradigm comes with multiple challenges on key aspects that haven't been solved yet, such as heterogeneity, standardization, security and discoverability. Most of the issues lie on the fact that the majority of mobile devices, which are easily accessible by any person, are able to be used as Edge nodes in certain implementations, effectively lowering the entry barrier to aiding the Edge. Thus contributing to the heterogeneity of the network and, as a consequence of that, standardization becomes a problem to a lot of devices, running different Operating Systems with disparate kinds of hardware, that might need to communicate with each other. Likewise, another issue resides on the security of the system, since ultimately any device will be allowed to contribute to the computation, where malicious users can perform eavesdropping and tampering attacks on the packets received. Finally, the discoverability aspect of Edge nodes and services in the network is very low due to the absence of a centralized point where costumers can connect to, in order to search for available services.

Furthermore, with the introduction of 5G technologies and smart bases stations which have processing capabilities and allow the offloading of applications' specific computing and caching strategies the station itself [11, 12], a whole new world opens up for the creation of highly performant and tailored mobile applications at the Edge.

1.2 Problem & Challenges

The problem that we are aiming to fix with the development of this thesis is focused on cases where a large number of users are located in the same geographical area and want to use their mobile devices to share and view, in real-time and in a persistent manner, generated content associated with their location. One of the most common scenarios are venue events, such as a football match in a stadium. Nonetheless, in this use cases, users often use cloud storage services, such as Dropbox [13], Google Drive [14] or even Instagram [15], to share their content. Although this approach works, it is not ideal since all the data transfers would have to traverse the entire backbone infrastructure to reach the cloud servers. Similarly, all of the other users that are interested in these items would also have to communicate with remote data centers, even though they are currently located right next to the content source.

Due to the high mobility, resource-limited, unpredictability, and out-of-order nature of edge devices and their respective over-the-air communication channels, creating a shared storage system spanning a wide geographical area that is simultaneously correct, energy conscious and performant is not a trivial task. This becomes particularly intricate in large capacity venues, where attendants are expected to consume and produce high volumes of data.

Frameworks and algorithms specifically designed for wired and static actors are not ideal for this new kind of network specification due to the intrinsic low tolerance to network congestion. Thus, replication and update schemes borrowed from those wired systems that tend to require a high message exchange ratio between clients, along with a packet order integrity, would not be useful nor efficient.

Multiple peer-to-peer edge-tailored storage architectures have been introduced to allow the creation of collaborative storage systems in bound-restricted geographical areas, such as Thyme [16–18]. This particular solution presents a novel approach to Publish/Subscribe systems for inherently mobile devices where the publish and subscription brokers are the clients themselves, which was proven to be very effective for data dissemination in edge networks, due to the interactions' loosely coupled nature and scalability. For instance, a framework like Thyme is able to establish and maintain a "shared folder" within a university department for students to share class notes or even photos. Consequently, if we wish to augment the created folder by linking more shared folders, and their content, generated in other distant departments with the goal of achieving one end-to-end shared folder, Thyme is unable to do so in scenarios like this since it relies entirely on P2P communication means, which are limited in range, as those departments could be located too far away from each other.

Furthermore, as a consequence of the novelty of this paradigm, a major problem lies on the fact that the research in this area, although growing in the last years, is still in its infancy without any kind of standardization from a third-party entity with regard to the offered services, protocols and privacy concerns. Therefore, the main challenges lie on the following topics:

- How to use infrastructure resources to increase performance and availability?
- What should be cached in the infrastructure nodes and in the clients' mobile devices?
- How to disseminate data to neighboring replicas as well as clients that are located farther away?
- How to guarantee that nodes will be able to continue to interact and view a consistent state of the storage system even without infrastructure?

1.3 Proposed Solution

For this thesis, the proposed solution is a collaborative storage framework, built on top of Thyme, specifically for Android [19] devices, with the goal of achieving a reliable and persistent infrastructure without the backend of a cloud service. By creating an open library, we intend to provide a simple and easy to use interface to allow any application developer to incorporate a fully collaborative storage scheme in their software.

We propose an extension to Thyme to allow utilization, when available, of previously deployed and stationary infrastructure nodes to aid the generated system, in terms of storage, logic and routing to other populated areas inside the same venue where using regular off-the-shelf wireless protocols would be infeasible or underperforming. Thus, nodes within the same *local area* (represented by the Basic Service Set of an infrastructure AP) can freely retrieve data objects. Additionally, each node can obtain items *globally*, i.e. from one *local region* to another, via the link provided by the infrastructure (Fig. 1.3).

As a consequence of the inclusion of infrastructure resources into the system, several major additions are employed by our solution:

- An **Overhaul to Thyme** and its Publish/Subscribe scheme to include the ability to interact with the infrastructure layer in a way that is completely transparent to the user;
- A **Local & Global Retrieval Method** where each a node will have the ability to get items from the same local network through direct peer-to-peer communication but will also be able to ask the associated AP for help to retrieve matching/relevant items from other locations;

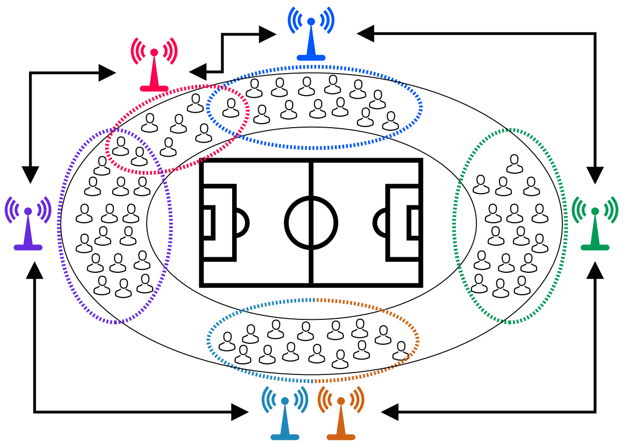


Figure 1.3: Example of a materialization of the proposed solution in a context of a football stadium. Nodes inside an AP's BSS form a fully complete local collaborative network but are also able to obtain and influence data items from other areas of the same venue via the AP's link.[1]

- A **Complex Caching Mechanism** employed by the infrastructure node to store not only the local items that are considered popular by its mobile clients as well as items originated in other venues areas that are of interest of its clients;

- A **Popularity Ranking Algorithm** to decide which items should be offloaded to the infrastructure caches for an higher availability.

Finally, our solution will be decoupled from the infrastructure, meaning that if an Access Point unexpectedly terminates or if none are available near the client, the system will continue executing normally.

1.4 Contributions

First, we develop Thyme-Infrastructure, a novel approach for infrastructure nodes in edge storage systems, such as access points or base stations, which is composed of complex caching mechanisms, structures and algorithms, linked together with the processing and routing capabilities built right in. Furthermore, this system connects users within the coverage of one node to share and obtain data from other infrastructure nodes' clients in the same venue, essentially establishing an end-to-end storage system to all clients, independently of their location. To achieve this, we propose a custom implementation of the Publish/Subscribe paradigm to be deployed on these low-resource infrastructure nodes, when compared to cloud systems, that orchestrates the entire dissemination workflow from one area to another with only the data that is relevant to the users from that area.

[1] Image contains icons made by Vignesh Oviyan, Freepik and Gregor Cresnar. Licensed under CC 3.0 BY.

Second, we update the Thyme mobile framework to search, make use and interact with the nearest infrastructure nodes running the Thyme-Infrastructure software stack. As described, Thyme employs a P/S scheme to aid its data discovery and dissemination through peer-to-peer interactions. With the addition of resources from the infrastructure, we amplify that particular Publish/Subscribe implementation by extending each P/S operation, that would have been only disseminated to peer nodes, to also be transmitted and executed in the infrastructure, such as the subscribe and the download operation.

Third, we perform an extensive performance study and evaluation of the proposed system in both real-world environments, using smartphones running Thyme and laptops to simulate infrastructure nodes running Thyme-Infrastructure, as well as simulated ones. In the case of simulated scenarios, we showcase how the entire system operates during diverse workloads and parameters. On the other hand, in real-world scenarios, we focus on gathering and exhibiting metrics like the overall battery consumption and transmission latency throughout the system execution. Further, we expand our evaluation suite by directly comparing our system with the previous specification of Thyme and with a cloud infrastructure.

Fourth, we highlight the proposal of other novel paradigms that we believe to be research-worthy to the entire scientific community.

1.5 Document Outline

The remainder of this document is organized as follows. In Chapter 2 we introduce the major areas of interest regarding the context of this thesis and state-of-the-art implementations of technologies and frameworks from those areas. Then, the Thyme project, which is the building block of the proposed solution, and its features are presented in Chapter 3. In Chapter 4, we describe a deep dive specification of the proposed solution, along with some implementation details. Consequently, we present a performance evaluation of our system, as well as a direct comparison of the system with the old version of Thyme and with the cloud, in Chapter 5. Finally, to conclude this thesis document, in Chapter 6 we raise some conclusions we gathered throughout the development of our framework as well as some future work that we believe would greatly improve the proposed system.

Chapter 2

State-of-the-Art

The purpose of this chapter is to present related work, concepts and implementations of multiple technologies and systems of computational and storage offloading in edge networks that are relevant to the development of this thesis. In order to successfully develop the proposed solution, most of the research was focused on specific areas of interest related to Storage Edge Networks in a peer-to-peer fashion, where mobility and context-awareness are crucial. Simultaneously, since we want our protocol to be as flexible as possible considering the characteristics of the network, systems where all the decisions are taken by the participant nodes themselves or by a local central node were also taken into account during each phase of the research. All of the researched frameworks that utilize these paradigms are fully described in Section 2.1. To finalize this Chapter, in Section 2.2, we will present some final remarks and takeaways from this research.

2.1 Collaborative Storage in Edge Networks

Regarding the collaborative storage aspect, for this thesis, we are especially interested in researching state-of-the-art implementations of systems created for edge networks, with a focus on mobile devices rather than static nodes. Furthermore, we want to have a technical overview of the most recently proposed solutions on where to place and retrieve data at the edge of the network, what should each client cache and when/how should the data objects be updated given the properties of the nodes themselves and the network.

Although our focus lies on mobile devices to create a totally collaborative and independent storage infrastructure, some of the architectures presented in this section are related to networks composed of mostly stationary devices or extensions to already developed cloud solutions where some of the decisions are actually sent to the cloud to be processed. The reason for this is related to the novelty of the edge and the associated mobile edge concepts, where these areas are rather unexplored and are yet to reach their full potential. Therefore, despite the fact that communication patterns and network stability tend to be widely disparate when comparing to highly mobile peer-to-peer networks, we consider that it is crucial to approach and study different solutions to learn and improve on, due to the absence of a standardized edge storage protocol elected by the research community.

Thus, we have the following groups of state-of-the-art implementations:

- **Data Sharing at the Edge for Mobile Clients**: Propagation- and Mobility-Aware D2D Social Content Replication [20], EdgeBuffer [21], Mobility-Aware Caching in D2D Networks [22], Peer Data Sharing [23], Nebula [24], SDMEC Storage [25], MECCAS [26], HDUM [27] and REAP [28];

- **Creation of Cloud Infrastructures at the Edge**: PopPub [29] and CoPro-CoCache [30].

2.1.1 What to Cache

A fundamental question in storage networks, specifically the ones located in low resource environments — when compared to a cloud corporate-grade infrastructure — like the networks located at the edge, is what should be stored and cached by the participants and/or the infrastructure in order to maximize the cache-hit ratio to a point where it is not detrimental to the overall performance of the network.

2.1.1.1 Popularity-based Approach

One of the simplest approaches, but the most common one, to decide which of the data objects should receive a high priority and become widely available through caching is the **popularity-based** approach [20–22, 29, 31].

In Propagation- and Mobility-Aware D2D Social Content Replication (PMAD2DSCR) [20], the authors define a novel device-to-device replication strategy through information obtained via social graphs. The proposed solution allows edge peers, in multiple regions, to cache data items based on estimates such as:

- *Inherent Content Popularity*: the estimated popularity of a given file taking only its content into consideration;

- *Influential Popularity* states that, after a content is published over social connections, its popularity will be determined by the social networks (shares, interactions, etc) and, as a consequence of this, the influence of each user over the entire network also needs to be measured;

- *Regional Migration Model*: an evaluation of how popular each file is with respect to the regions (since different users in multiple geographical areas could lead to different interests) and the migration ratio between each pair of regions;

- *User Regional Mobility Prediction*, which represents the probability that a user will move from one region to another after a given time slot.

Each edge node will be assigned to replicate, in an on-demand manner, a subset of all the available content to its peers based on the above popularity-based estimates that it was able to calculate with the information that is locally available. Those contents could be retrieved by directly receiving them from other nearby edge nodes (which will need to notify the coordinator server for it to be aware of the locations of each file within the entire network) or by receiving them from the content server. Additionally, when a user wishes to obtain a copy of a particular file, it will first inquire the coordinator server to find which users in the same region have replicated the content and can randomly choose one to download the content from.

Similarly, EdgeBuffer [21] is a caching and prefetching method, culminating from the collaboration between geographically distant access points, that also uses the popularity of each data and the prediction of users' mobility as the main criteria to choose the best cache strategy. In this system each AP maintains a table and, for each entry (i.e. a block of data), associates a counter which represents the number of times that particular item was accessed by other nodes. Whenever the AP's cache reaches the total amount of available storage space and an incoming storage request is received, if the lowest item counter in local storage is lower than the received counter, that item will be swapped by the new one. By performing this on-the-fly disposal and update process, the designed framework guarantees that only the most accessed items are effectively stored and retrieved by the clients.

PopPub [29], a lightweight distributed collaborative caching strategy, is conceptually equivalent to EdgeBuffer, where nodes leverage the popularity aspects of each data to make caching decisions, but this framework is developed for a particular kind of network, such as Content-Centric Networks [32] — a specification of the widely deployed Content Delivery Networks [33] — which is a novel approach that utilizes various Edge Network concepts and properties, such as the caching and routing collaboration of infrastructure nodes. In CCNs, broadcasted messages lie in two specific types: interest packets (which are essentially request messages) and data packets (the response, containing the data, to an interest packet). Upon processing a data request, the content sources then transmit the data packet back to the client along the same path as the interest packets.

In order to compute the popularity of each item, the authors propose that every time a low traffic edge node receives an interest packet to be routed, it also maintains and counts the request rates associated with that item. By limiting this functionality to only the routers with the lowest workload, the most active nodes are not impacted by a performance downgrade as a consequence of the additional step of incrementing the value, and caching redundancy is minimized.

To transmit the popularity information (Popularity Information Base, PIB) of a content to other nodes, instead of periodically broadcasting that information network-wide — which would culminate in a huge communication overhead —, that knowledge is only published to the nodes along the interest packet forwarding path. That is, edge routers update their PIBs upon an interest packet arrival and other routers, such as aggregation and core routers, update their PIBs upon a *pub* packet, which contains multiple data items grouped together — based on the original transmission path — and their popularity counts in one single message. Thus, during a packet forwarding operation, each router checks their local PIB and decides whether the data packet is popular enough to be cached locally or simply routed back to the user.

The Pattern Based Popularity Assessor (PBPA) [31] — although not a full-fledged edge framework — is a ranking algorithm especifically designed to be deployed on top of edge (storage) systems with the goal of determining and predicting the popularity of mobile files to, ideally, be offloaded. To achieve this, Elgazar et al. studied and analyzed real-world access traces of several different mobile device users, and multiple file types (such as video, audio and image), in order to gain knowledge about access patterns at the edge and construct a formula tailored for the necessities of these clients.

With the authors' findings, the proposed algorithm is composed of the following steps:

1. Divides accessed files into two file sets: files not accessed in the most recent time period (stale files), and files accessed in that period (active files);

2. Sorts the active files by their number of accesses, and considers the top portion of those files as popular;

3. Analyzes the clustering patterns of the selected popular files, and among the remaining active files it deems those who have similar access patterns to be popular as well, while the remainder would be unpopular.

Finally, the solution proposed in Mobility-Aware Caching in D2D Networks (MACD2DN) [22] brings new additions to popularity-based caching algorithms by fully taking advantage of edge users' mobility-aware characteristics, such as movement patterns and context-awareness' inter-contact times between different users (the time that they can contact with each other and, possibly, exchange data), in order to compute a caching strategy to maximize the data offloading ratio (percentage of data transmitted through peer-to-peer interactions than through base stations).

The authors then define a dynamic programming algorithm to achieve the proposed near-optimal caching performance and placement by taking into consideration some observations from real-life data-sets and interactions, specifically: users that tend to move slowly within the network tend to cache the most popular files to indulge their own needs, while fast-moving clients tend to cache popular files as well, but with the purpose of fulfilling requests from other users, since more opportunities to communicate with other peers are presented. On the other hand, medium velocity users showed that they cache less popular files to take advantage of device-to-device communications.

2.1.1.2 Metadata

Another approach to caching data in a collaborative storage environment lies on the creation and maintenance of **metadata** information, which tends to use a fraction of the size of the original data, and is used to broadcast knowledge to the entire grid without the performance overhead caused by the transmission of messages with a considerable amount of bytes. One example of an edge solution using metadata is Peer Data Sharing [23], which refers to a model where edge clients can dynamically discover which data exists in nearby peers' local storage and, from all the items returned from the network, choose which one(s) he/she is most interested in.

Song et al. refer that whenever a node is interested in generating a new data item to be accessed by the entire network, a data descriptor (i.e. metadata) is created and associated to it which consists of multiple attributes such as the name, the GPS coordinates where it was originated and, if the data represents a large item, the total number of chunks.

In order to obtain a previously created piece of information, the requesting node needs to trigger the *get* operation by broadcasting a query message to the network, specifying which data items should be retrieved: each node that receives this request should reply to the original node the list of all metadata items that it currently holds. During this entire process, whenever an edge device receives, relays or overhears a message containing the metadata entries from a neighboring node, it will also cache this information to serve potential future requests. Thus, nodes, including the one initiating the request operation, will be smart enough to be able to make best-effort routing decisions to forward the messages in the directions where the requested data should exist. On the other hand, if such information is not present (a newly formed network or a node that recently joined it), a flood operation should be performed in order to obtain an overview of what was already shared.

After receiving the initial round of metadata items, the consumer dynamically decides whether and when to start a new round or terminate the data discovery if it determines that almost all data entries are returned. If, for instance, the client considers that the information received is not enough, he/she is able to explicitly tell, during the next broadcast, which nodes that already answered successfully and should be ignored, minimizing the number of packets sent.

Having all the information needed, the node broadcasts a message asking which of its closest neighbors contains a block of data, corresponding to the requested item, and performs a *chunk query* to each one of them.

2.1.1.3 Discussion

Table 2.1: "What to Cache" approaches comparison.

	Predominant Cached Data	Uses File Metadata	Caching Decision	Data Request Model
[20] PMAD2DSCR	Whole file	No	Popularity-based	Pull model
[21] EdgeBuffer	Whole file	No	Popularity-based	Pull model
[29] PopPub	Whole file	No	Popularity-based	Pull model
[31] PBPA	Whole file	No	Popularity-based	N/A
[22] MACD2DN	Whole file	No	Popularity-based w.r.t location & velocity	Pull model
[23] Peer Data Sharing	Metadata	Yes	N/A: All metadata are cached	Pull model

In Table 2.1 we show a focused comparison of the previously detailed approaches regarding the "what to cache" primitive. As we can observe, the majority of the researched systems employ a whole file-only paradigm for their caching mechanism. While this approach is clearly the simplest to implement, we believe that only relying on this methodology would not be ideal for our solution: having to constantly transmit files with a considerable size, in the case of photos for instance, from one node to another can have huge implications in such networks as the ones we are considering for this thesis, specifically due to the congestion and unreliability aspects already mentioned. Therefore, we intend to combine the usage of both whole file and metadata information, where the latter will be predominant throughout the network whereas the former should ideally be used on the nodes that actually wish to receive the data object.

Regarding the caching decision aspect, it is also possible to verify that the bulk of edge storage systems use some kind of assessment to choose what should be cached and available for the rest of the network, particularly popularity-based estimates. As described in Section 1.3 we are interested in exploiting infrastructure nodes to support the network storage and routing requests. Thus, implementing estimates like the ones used in the PMAD2DSCR project, such as the *Inherent Content Popularity* and *Influential Popularity*, and in the PBPA algorithm could be proven highly useful for dissecting the shared data items and select the top *k* objects to be stored on each infrastructure node, based on each BSS local traffic, for easy access and discoverability.

In terms of the data request model, most of the recently proposed collaborative edge systems use an on-demand client-by-client pull model, as we can corroborate with the findings in Table 2.1. Since we are using the Thyme framework, a Publish/Subscribe system, as the base communication protocol for our collaborative storage network, we will effectively use a mix of both push (when nodes receive a broadcasted notification that a relevant item has been created) and pull models (after receiving the content notification, the client will then individually inquire a replicating node for to retrieve that data).

2.1.2 Where to Place the Data

As with the decision of what should be stored in the network, whether it is the original data or a representation of it, estimating where the actual items should be stored has an equally important role on Edge networks. In this type of system, it is crucial to take into consideration the volatility and inconsistency of the over-the-air communication channels used at the edge, which are highly conducive to congestion. Ideally, frameworks should be able to dynamically perform a best-effort cache placement decision to counteract the inherent issues presented above, by utilizing the nodes' location and possible congestion areas, with the goal of increasing data locality and minimizing the number of network requests when obtaining the object.

2.1.2.1 Origin-placement

The simplest approach to a problem of this nature is to use an **origin-placement** strategy where, in terms of effective availability for the rest of the network, the generated data does not leave the producer node's local storage. This methodology is employed by the previously referred Peer Data Sharing [23] model.

2.1.2.2 Reliable Nodes

An alternative implementation is based on offloading data to **reliable nodes** on the Edge infrastructure. As a consequence of the low entry barrier for devices to be able to collaborate on the Edge, any client, including ones with malicious intent, could easily connect and tamper processing/storage procedures. This particular behavior would not be a major concern in a cloud service context since the administrators would have full control over where data would be stored and available, by using trusted servers. A materialization of this particular approach is performed in Nebula [24].

The Nebula system is composed of multiple services/actors, such as the *Nebula Central* (central node which acts as an entry point for other nodes to join the infrastructure), the *DataStore* and *ComputePool* (set of edge devices that specified to the *Nebula Central* that they would like to contribute with storage and processing, respectively, or both) and the *Nebula Monitor* (which acts as a supervisor that monitors the performance and the characteristics of each volunteer node).

On the other hand, it might also be crucial to provide dedicated nodes in addition to the volunteer devices, by the infrastructure administrator, due to reasons like privacy concerns, where owners of the generated data might only want to explicitly save them in trusted dedicated nodes, and the necessity to save items in locations that provide higher reliability, since volunteer-based nodes tend to be less reliable and/or have a direct contribution to the overall churn of the system. To manage each of the data nodes, the *Nebula Central* itself or an elected *DataStore Master* keeps track of all active devices, the files metadata, file replication information and the data placement policy.

The proposed infrastructure also provides load balancing, with the help of the *DataStore Master*, by logically ordering all nodes by the percentage of their occupied storage, so that new data can be stored in a data node that has the lowest load, and locality awareness by maintaining a data structure with the effective bandwidth between each pair of nodes. This way, whenever a node wants to determine which other node has the highest locality to a given data item, it will send a query to the *DataStore Master* and the maintained data structure will be returned, in an ordered manner.

2.1.2.3 Infrastructure Offloading

Other frameworks have been implemented to take advantage of previously built and deployed **infrastructure** nodes, by preferring to save data in those places over other customer devices, which is the case of SDMEC Storage [25] and EdgeBuffer [21]. These particular nodes usually have higher processing and storage capabilities and are more reliable than regular mobile gadgets.

The authors of SDMEC propose a cooperative Mobile Edge Computing storage architecture for MEC nodes consisting of three layers: the Data Layer (the storage infrastructure containing the assets), the Control Layer (which is responsible for managing and controlling storage resources - Storage Controller - and controlling the network requests and access policies - Network Controller) and the Application Layer (which interacts with the MEC services).

The system allows any node inside the radio coverage of a given MEC to store some data of a specific size. Before starting the store process, the local MEC's Storage Controller verifies if has enough storage space to fully serve the node's request. If so, the data is successfully stored in the MEC. Inversely, if the local MEC is incapable of handling the storage request due to a shortage in size of the available resources, it stores all possible blocks of data locally, while the remaining blocks are transmitted to other neighboring MEC nodes, searching from closest to farthest in order to maintain efficiency, until all chunks are stored.

One downside of this approach resides on the possibility that all of the closest MEC nodes could also be full and unable to process the storage request, leading to cases where only the farthest nodes could respond to the request. As a consequence of that, chunks of the same file would be scattered across a considerable geographical area, having a negative impact on the performance of a *get* operation. To overcome this, the framework gives the MEC node the right to perform an auto-scale request to the infrastructure and increase the storage capacity of each MEC if the described search process surpasses a given threshold of MECs inquired.

Additionally, EdgeBuffer [21] — introduced in Section 2.1.1 — incorporates the sense of **mobility-aware prediction** to the above methodology by forcing each infrastructure AP to assess, after a client performs a join operation, how long will it stay in the same Basic Service Set and which will probably be the next access point it will connect to. To do this, each access point has a private table with items for each user, containing its last AP, the residence time in that base station and the next AP it moved towards to. Furthermore, during the user's transition between one location to another, the current AP will forward the contents of its private table so the next actor will be able to estimate and transfer the blocks that the user will demand during its expected residency duration.

2.1.2.4 Offload to Neighbors based on Hardware Information

Another novel solution for the data placement problem, proposed by the authors of MECCAS (Mobile Edge Cloud Collaborative Storage) [26], leverages the **hardware information** of each neighboring node, with regard to the intrinsic characteristics of the actual published data, to dynamically decide which of the clients will be assigned to store the objects. This is performed with the goal of minimizing the delay of task execution and total costs for the overall storage operation, while maximizing the utilization of local information of nodes and system reliability.

In the MECCAS architecture, the edge network is composed of request and storage nodes, where the former are the devices that will eventually ask the system to store data and the latter being the clients that will only lend storage resources to the entire infrastructure. Furthermore, nodes can be of type "request" and "storage" at the same time.

In order to avoid distinct criteria to measure the size of a storage request, which could be highly biased to the hardware of a device, the authors introduce a *basic unit*, which is comprised of multiple elements such as the *storage capacity*, *standby time* and *bandwidth*, that is made to define how much resources a storage node will need to execute a request. With this information provided by the neighbors, the requesting node will then calculate the optimal way of storing its data within the network.

Additionally, requesting nodes also need to consider other constraints to make their decisions as solid as possible, namely: *system reliability*, defined by the failure rate of each node, the probability that the same node will be restored in case of storage failure and the risk associated with the request itself; *power usage effectiveness* which takes into account the power usage of each storage node at idle and peak states; and the *risk of nodes withdrawal* in order to avoid allocating tasks to nodes with higher probability of leaving the network. Since these particular constraints are critical to maintaining a healthy network state, they are constantly being updated.

2.1.2.5 Along the Routing Path

The formerly introduced popularity-based cache framework, PopPub [29], also implements a placement strategy, which differs from the rest of presented proposals, by storing the data **along the routing path** of the request. Since in CCNs there are multiple content sources for different topics, the creators discuss that by caching the objects on the nodes along the way, between the requesting client and the source, will lead to better results due to the fact that each particular item will only be present in one of the available content sources in a given geographical area. Thus, independently of the user that requests the object, the routing path will tendentially collide with the majority of routes formed from requests of other nodes in the same area.

2.1.2.6 Discussion

Table 2.2: Data placement strategies comparison.

	Local Nodes' Storage Usage	Uses Infrastructure Storage	Data Placement Strategy
[23] Peer Data Sharing	Origin + Distributed (metadata)	No	Origin-placement
[24] Nebula	Distributed	No	Reliable Nodes + Volunteers
[25] SDMEC Storage	—	Yes	Nearest Infrastructure Node
[21] EdgeBuffer	—	Yes	Mobility-Aware Prediction
[26] MECCAS	Distributed	No	Neighbors' hardware information
[29] PopPub	—	Yes	Along the routing path

One major takeaway from the data placement strategies comparison in Table 2.2 is associated with the fact that fully collaborative storage systems that utilize both local nodes' storage plus infrastructure base stations', in the sense that all standard actions are available — create, delete, get and update — in both places, is still widely unexplored. Although none of the studied systems present a similar infrastructure to what we are trying to achieve in Section 1.3, there are certain strategies employed by the above frameworks that could be particularly interesting to implement in our solution, such as Peer Data Sharing's origin-placement, Nebula's reliable nodes and PopPub's caching along the routing path.

With the origin-placement, we believe that maintaining the produced data object in the source node has multiple benefits, particularly by eradicating the need to redirect the entirety of the data to another place inside the network and, most importantly, location-awareness. By taking into consideration real-life interactions, the latter paradigm is guaranteed by keeping the generated data at the origin since neighboring users will have a high probability of being interested in receiving the content that is shared in their local geographical area, e.g. within their range, which also minimizes the requesting messages' travel time.

By utilizing infrastructure equipment in our proposed solution to aid and collaborate with the surrounding edge devices, it is possible to effectively see them as "reliable nodes" since they were provided by the network engineers in the current venue. Thus, such nodes can be considered reliable with respect to their stability, e.g. will not shut down due to shortage of battery and will be able to effectively handle multiple requests at the same time, but also to their impartiality over privacy concerns regarding data that is shared in the network. Therefore, it could be interesting to take advantage on-the-fly of these notions and improve the network's quality of life. One example would be to automatically offload data that are associated with a particular location where the majority of the nodes are leaving, meaning that the local files could be effectively lost as a consequence of the churn, to the nearest base station in order to guarantee data longevity.

As shown in Fig. 1.3, if the current network presents a similar topology, i.e. infrastructure nodes are deployed throughout the geographic location of the venue and be directly or indirectly connected to each other, it is our belief that exploiting the connectivity and routing ability between those static nodes to place specific data items on the base stations along the routing path of requests, such as the most popular ones, could be very effective in the long run and decrease the round-trip time of requests to those objects.

2.1.3 How/When to Transmit Data/Updates

One of the most important aspects in edge networks, when considering data updates, is that the process should be as simple as possible without the need of a globally strict replication and consistency scheme due to the high volatility of communication channels used by nodes near the edge of the network. Furthermore, since edge devices are also expected to be unpredictable with regard to their mobility and needs, updates to shared data objects should be applied in an opportunistic manner whenever possible, with the goal of achieving high performance while minimizing the number of messages to be flooded through the mesh in order to achieve a consistent view of each item.

During this research, one of the challenges faced was tied to this particular topic since most of the proposed solutions do not offer data updates at the edge or, if this action is indeed offered by the system, it is a full overwrite function masked as an update, which is not ideal in this type of ecosystem since the entire data needs to be sent whenever an update action is triggered.

The following frameworks represent the most interesting approaches, found during this research, related to the transmission of information.

HDUM [27]: one example of an edge-optimized solution is the Heuristic Data Update Mechanism, an opportunistic data update mechanism for peer-to-peer edge networks. In this proposal, the authors note that the network itself is partitioned in multiple distinct geographical groups which contain various Member Nodes (*MN*) and at least one leader (the *Group Node*, *GN*), and the latter is elected, among all local *MNs*, based on its storage capacity, bandwidth, computing power and retention time.

Periodically, each *GN* broadcasts a beacon messages with a limited hop count, in order to collect all the shared data items in its group. Thus, when a *MN* receives that beacon packet it automatically determines which of the groups it belongs to. Although this approach is conceptually and programmatically very simple to implement, it bears major drawbacks due to the possibility of over flooding the networking with the routing messages. One solution proposed by the authors is to maintain a *holder list* in each of the nodes, which contains the set all of the neighbors that currently store a particular data item, that needs to be updated from time to time. Nonetheless, another issue arises in alternative modern solutions when updates are performed in distinct areas of the same infrastructure, as a consequence of all the messages that need to be sent throughout the entire network in order for each node to maintain a consistent view of the data that it currently holds, resulting in a waste of bandwidth and performance detriment.

In order to minimize the number of messages transmitted, the proposed solution combines two distinct approaches:

- **Safe-time Derivation**: temporal prevision of when a neighboring node will leave its radio range and become disconnected, in order to schedule the best time (*safe time*) to update and verify the consistency of its data with respect to the data stored in the disconnecting node. To achieve this, each node exchanges the location and speed/GPS information of itself with its neighbor nodes when it joins the network.
- **Relative Ability Filter**: best-effort estimate of a given node's stability (*relative ability*), using the gathered information such as its network connectivity, access frequency to a specific data item by that node and its residual battery capacity (this last one was considered to be the most important by the authors). That is, if the *relative ability* of a given node is below a certain threshold (meaning that the device might be leaving soon due to its battery running out, or it cannot spread data efficiently or it scarcely accesses the data item, so it is not urgent to update its local copy) it will not receive any data updates.

By combining the above estimates, HDUM is able to optimize the update process by only choosing to execute it if it indeed is worth the network overloading and by choosing the most convenient time to perform the action. This proposal is particularly interesting due to its novelty, but also because it fully leverages the mobility & context-awareness characteristics of edge devices.

REAP [28]: on the other hand, REAP is another reliable and efficient ad-hoc network protocol that leverages the Publish/Subscribe mechanism to notify edge nodes on data submissions through means of delta compression. When a REAP producer transmits a new message to the network, it starts by checking its cache containing previously published documents. Then, the MultiDiff algorithm proposed by the authors identifies matching character sequences between the stored documents and the new one, and, by choosing the best subset of documents for the matching algorithm, creates the smallest patch possible that needs to be applied on those documents in order to generate the one that was recently created. If no relevant character sequences were found, the document is transmitted as a whole after applying a zlib [34] compression pass.

When a patch is successfully generated, meaning that it references one or more documents as well as the new data content, and received by a subscriber, due to the out-of-order nature of edge networks the client may need to wait for a few moments if he/she does not have some of the documents referenced in the patch. Thus, in this case, the patch will need to be momentarily stored on the side until those absent files are received by the user for the patch to be retried.

The key mechanism in the REAP protocol lies in the decision of how many and which documents should be cached in order to be referred inside patch messages. The authors propose two distinct approaches: a time-bounded or size-bounded window. In the first specification, the producer's MultiDiff instance will attempt to use the documents sent within the last k seconds to calculate a patch. For instance, if a window of 60 seconds is specified, a full document will only need to be sent every minute. Therefore, if a new document is published every second, only the first packet will contain the full bytes and the next 59 will contain small patches. On the other hand, when using the latter approach, the producer will only be able to utilize the last n messages sent for the diff. Nonetheless, independently of the implementation used, a small number of bytes are associated with each packet to inform the consumers when to flush their document caches.

We considered this specification to be worth studying due to its similarity with our proposed solution, specifically the Publish/Subscribe module. The fact that this framework utilizes deltas, that refer to previously sent files, in order to build a newly created object is a rather compelling approach to greatly decrease messages size. In spite of this, we recognize that the authors' approach is highly vulnerable to packet loss and would not perform as well on storage networks where the published content structure and specification is widely disparate. By taking into account the latter, the algorithm would not be able to create patches, falling back to a compressed whole file transfer in most of the cases, which would be inefficient.

CoPro-CoCache [30] is another implementation of a cloud service at the edge, materialized as a framework that allows infrastructure nodes to collaborate on video streaming caching and processing. The proposed solution is based on a single MEC server in each location that acts as a caching and transcoding node. Furthermore, the servers keep track of the most popular videos and each of them is evenly distributed among all the APs in a predetermined bitrate quality (tending towards higher bitrates). However, when a user asks one of those videos but in a lower quality, whether it is preferable due to network congestion, low battery percentage or diminished processing capabilities, the associated MEC will be responsible for applying an on-the-fly transcoding function to respond with the desired quality. The node is able to do this by checking if its local storage contains the requested video or by asking its neighboring MEC servers, without the need to communicate with the cloud infrastructure and obtain a new video file for the given quality.

One of the major takeaways from the CoPro-CoCache specification that we acknowledge as an asset to our system lies on the idea of being able to adaptively create smaller representations, in size an quality, of a data item. Furthermore, we could implement an iteration of this proposal in our solution to place specific quality-diminished data items (e.g. based on popularity) inside each infrastructure AP. This approach would be able to greatly increase system responsiveness on requests for those particular objects, while, at the same time, having a mechanism that would eventually send the object in its original quality to the user, in a later, more opportunistic, time frame.

2.2 Final Remarks

Throughout this chapter we performed a survey related to state-of-the-art systems and implementations that we consider relevant for the development of our proposed solution. With this research, we were able to gather insights on solutions for novel obstacles related to Edge networks, a broad area that still remains fully unexplored. In the next chapter we will introduce another state-of-the-art edge framework, Thyme, which will be used as the foundation for our specification. Then, in Chapter 4, we will present a detailed view of the framework that we are proposing which aims to solve the problems introduced in Section 1.2.

Chapter 3

Thyme

In this chapter we present the Thyme system, starting with a technical overview in Section 3.1, followed by in-depth description of each of its features, like the publishing of data, replication procedures, subscriptions and data retrieval, mobility-awareness and mitigations in Sections 3.2, 3.3, 3.4 and 3.5 respectively.

3.1 Overview

Thyme [16–18] is a topic-based time-aware publish/subscribe system, tailored specifically for mobile edge networks, which forms a persistent collaborative peer-to-peer storage ecosystem. In this scheme, users are able to specify, on their subscriptions, whether they wish to receive items that were published in the past, data that is currently being created or be notified of future topic occurrences. The authors note that Thyme is the first P/S interface with support for subscriptions within a time scope that can refer to the past.

As with regular publish/subscribe systems, this novel T-P/S approach also offers the publish, subscribe, subscription cancellation and previously published data retrieval operations. Furthermore, since Thyme considers time to be a first order dimension, additional custom operations were added to this framework, such as the ability for a node to unpublish a previously shared data, effectively deleting it from the shared storage, since it would not make sense for a future node to able to retrieve a data from the past that was deleted. On the other hand, modern P/S systems offer limited subscribe functionality capabilities by only allowing each client to state a single topic keyword per subscription, whereas Thyme supports a more complex set of features by allowing logic formulas with topics/keywords as literals.

In Thyme, nodes are considered to be functionally symmetric, sharing the same responsibilities and having no particular roles, meaning that there are no centralized or specialized components, and each node can be a publisher, a subscriber, or both. Moreover, in this approach, the geographical space where the system lies is logically divided into multiples cells with equal size, where all nodes inside a given cell collaborate with each other to form a virtual node. To fully take advantage of this network layout, the framework also uses a geographic hash table (GHT) to check which logical cell is responsible for managing and storing the published objects' metadata for a given tag.

This framework was developed in the context of the *"Hyrax: Crowd-Sourcing mobile devices to develop edge cloud"* project [1] and acts as a foundation for the purpose of this thesis.

3.2 Publishing Data

For each data object that is shared by the users in the network a new metadata item has to be created, associated to it and transmitted by the author, which consists in a tuple $\langle id_{\text{obj}}, T, s, ts^{\text{pub}}, id_{\text{owner}} \rangle$ where:

- id_{obj} acts as the object's identifier;
- T is a set of tags or keywords related to the object, e.g., hashtags used in social networks;
- s is a summarized representation of the shared object, e.g., a thumbnail of an image in the case of a photo sharing network;
- ts^{pub} is the object's publication timestamp to guarantee temporal perception, used to bootstrap the time-aware functionality of the developed solution;
- id_{owner} is the publisher's node identifier.

When a publish operation is executed, the owner's Thyme instance leverages the information provided by the GHT and, for each keyword contained in T, sends the metadata information to the cell that is mapped by the hash of each tag and every single one of its inner nodes will permanently store the given tuple. On the other hand, the actual data object stays stored in the owner's node, ensuring that only the metadata is sent through the network, saving bandwidth and overall network resources since the metadata tends to be much smaller than its data counterpart. Figure 3.1 illustrates the process of the usage of the GHT as well as the selection of the geographical areas which will receive the data's metadata information given the tags chosen by the owner, in this case "beach" and "summer".

One shortcoming of the approached solution in Thyme is that objects are read-only, meaning that they can never be edited after the original publish operations.

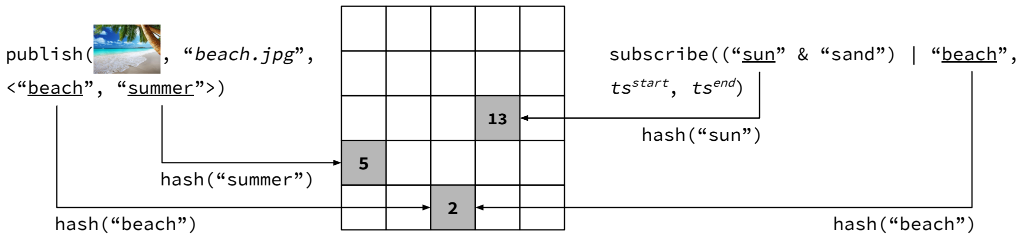

publish(, "beach.jpg", <"beach", "summer">)
hash("summer")
hash("beach")
subscribe(("sun" & "sand") | "beach", ts^start, ts^end)
hash("sun")
hash("beach")
13
5
2

Figure 3.1: Example of Thyme's publish and subscribe operations. The tags' hashing determines the cells responsible for managing the object metadata (cells 2 and 5) and the subscription (cells 2 and 13). If a subscription has overlapping tags with a publication (and vice versa) it will also have overlapping (responsible) cells, guaranteeing the matching and sending of notifications to the subscriber. Adapted from [17].

3.3 Replication

This framework offers two types of replication mechanisms to ensure that data is not easily lost, caused by the volatility of edge peer-to-peer networks:

- **Active Replication** leverages the GHT's generated clusters to replicated objects within the network. Upon a publish operation, the owner of the data disseminates the item to its peers inside the same geographical cell. By doing this, every node that received the data this way should be able to reply to data requests for that particular object, effectively maximizing the network's tolerance to churn, especially if the publishers leave;

- **Passive Replication**, on the other hand, enables nodes that are outside of the geographical fence of the original data publisher to replicate the item from the moment they download it. This allows the items to be scattered across the network, increasing data availability and performance since a node will ask the closest replicating device to retrieve a particular data.

In order to fully take advantage of the implemented replication methods, Thyme adds a new field to the metadata information specified in 3.2, the *replication list* L_{rep}, which contains the set of all nodes, independently of where they reside in the network, that are currently in possession of the data. As a consequence of this, the metadata is now composed by $\langle id_{\text{obj}}, T, s, ts^{\text{pub}}, id_{\text{owner}}, L_{\text{rep}}\rangle$, where L_{rep} is a list of pairs $\langle id_{\text{node}}, cell_{\text{node}}\rangle$.

3.4 Subscription & Data Retrieval

To perform a subscription operation, the requesting node needs to send a message to the network containing the subscription metadata in the form of $\langle id_{\text{sub}}, q, ts^{\text{s}}, ts^{\text{e}}, id_{\text{owner}}, cell_{\text{owner}}\rangle$, where:

- id_{sub} is the subscription identifier;

- q represents the query's logic formula which is allowed to contain keyword conjunctions and disjunctions;
- ts^{s} is the initial timestamp of published data to be retrieved, 0 if the users want to get all data from the beginning of the system related to the specified query;
- ts^{e} contains the timestamp's upper of published data to be retrieved, ∞ if the user wishes to receive all future data matching the query;
- id_{owner} represents the requesting node identifier;
- $cell_{\mathrm{owner}}$ holds the identifier of the logical cell where the requesting node is located.

Furthermore, the client's Thyme instance starts by parsing the specified query in the subscription message with the intent of verifying which of the network cells will receive the subscription metadata, using the GHT information, while trying to minimize the number of cells needed to process the subscription — which has a positive impact on the total amount of packets needed to be sent throughout the entire network.

After these steps are completed, the messages are then sent to selected cells' nodes and, from that moment on, they will be responsible for storing and maintaining the list of active subscription requests and notifying the requesting nodes whenever a new data item is published that matches the specified parameters, particularly the tags, ts^{s} and ts^{e}. Thus, when the notification messages containing the object's metadata are triggered, routed to the recipient clients and the users are interested in obtaining the item, each node will transmit a data retrieval message to the closest replica in the L_{rep} field (as depicted in Fig. 3.2). If, after a given threshold, no reply message arrives the requesting node, possibly due to the fact that the node replicating the object went offline or left the network, the client will be in charge of iterating the rest of the L_{rep} list, from the closest to the farthest replicas, until the data is successfully retrieved.

Moreover, after receiving the item, the current node will contribute to the passive replication for that particular object.

3.5 Mobility-Awareness & Mitigation

Since edge nodes are expected to have the ability to freely and spontaneously move inside the geographical space of the network, Thyme is built to embrace this and mitigate possible issues that could be caused by the mobile characteristics of the clients' nodes. To achieve this, whenever a device notices that it starts to move in a considerable way, Thyme broadcasts a warning message to its cell's peers stating that it is no longer capable of aiding the network at the moment, meaning that it will not be chosen to process client requests but will be able to continue receiving local update messages.

On the other hand, when the framework notices that the user is slowing down, transitioning to a stationary state, Thyme communicates with the neighboring nodes to return

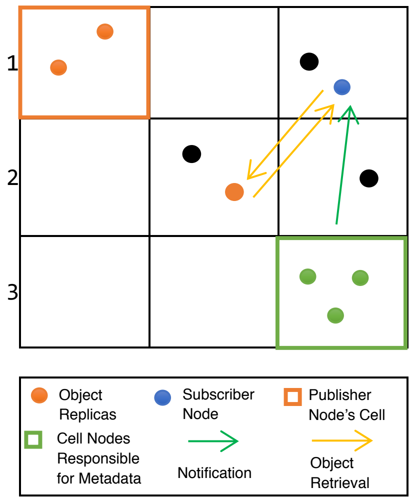


Figure 3.2: Thyme's subscription notification & data retrieval process. Adapted from [18].

to an available and working state. If the client remained in the same cell, no housekeeping is necessary and it is ready to contribute to the network. Otherwise, the transitionary node has to perform additional steps in order to harmonize its own and the entire network state. First, it needs to update its own state with the state of the entire cell in order to be able to process the cell's incoming requests. To do this, it communicates with one of its neighbors and asks its whole state.

Secondly, if the moving node is passively replicating at least one data object, it is necessary to inform the respective cells that index the metadata information of the items in order for the nodes to update the L_{rep} field, since the replica location has changed.

Then, for each active subscription, Thyme will also notify the cells that are responsible for processing the subscriptions and notifying the user on new submissions, in order for the inner devices to update their local state with the most recent location. This means that subscription notifications will not be transmitted or get lost in the previous location.

Finally, in case the current node has previously performed a publish operation on the network, the framework takes the initiative to transmit management messages to all of the cells that index and maintain the items' metadata information with the latest publisher location, also updating the replica locations as a byproduct of that. Consequently, the node will then be considered a passive replica of the data.

Those clients that decide to leave the geographical space of the network will stop receiving messages and be disconnected from the system.

3.6 Worlds

The authors of Thyme define "World" as the geographical space where the network operates, which is mapped by the GHT. Furthermore, the framework also allows multiple worlds to coexist in the same, or overlapping, areas without any kind of issues regarding

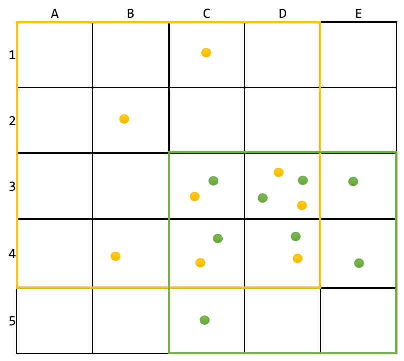


Figure 3.3: Overlapping Thyme Worlds. Adapted from [18].

synchronization and shared resources, although users can only be part of one world at a time.

When activated, Thyme allows the client to join an ongoing world or create a new one from scratch. To join one, the device first listens to the packets that are transmitted via the network (i.e. "hello" messages), for a predetermined amount of time, to check if there are active worlds to join. If no worlds were found, or if the user intends to start one, multiple parameters have to be specified, such as its geographical area, including the central point, its name for others to search and join, and how long should the world be active. At the same time, a unique identifier is also generated and the size of each cell is calculated based on the communication protocol that will be used inside the world (Wi-Fi, Bluetooth, etc).

In order to fully take advantage of overlapping networks, nodes from different worlds that are currently located in the same geographical space can collaborate to increase performance and availability. Thus, those nodes will be able to store objects and metadata from both worlds but are only allowed to reply to data requests from nodes that reside where the requested item lies (Fig. 3.3).

Chapter 4

Thyme-Infrastructure

In this chapter we will present, in detail, our proposed solution, Thyme-Infrastructure (subsequently referred as Thyme-Infra), including its model definition, architecture as well as a description of its core features. Additionally, some of the most relevant implementation details will be outlined. Throughout this chapter, we will interchangeably use the terms infrastructure node, Thyme-Infra node, base station (BS) and access point (AP) to denote an infrastructure hardware component that is running an instance of Thyme-Infrastructure.

4.1 General Overview

The solution that we have developed, in the context of this thesis, is a fully collaborative storage edge network infrastructure, built on top of the work developed for the Thyme system, allowing any mobile node to create, retrieve and delete any object inside the network without the need of an active cloud service to manage interactions and store data. Furthermore, we are exploiting the use of stationary nodes whenever possible or available, namely venues' base stations, that are located near the mobile clients. These base stations are deployed in a fully or partially connected mesh-like topology, through wired means (Fig. 1.3), and are able to aid the system in terms of storage, processing and routing. Therefore, by having infrastructure nodes take an active role in the edge network, it is possible for clients to communicate and interact with other nodes in a geographically distant place, although in the same venue, where using wireless off-the-shelf communication means, e.g. Wi-Fi Direct and Bluetooth, would be underperforming or even impossible due to range limitations. Thus, we are proposing an overhaul to the Thyme's mobile system in order to interact and utilize the resources within the infrastructure as well as the creation of a whole new software stack built in Java that will run on those infrastructure nodes, namely Thyme-Infrastructure, thus culminating in an intricate two-layered system. While both of these components make up the Thyme ecosystem, we will further call Thyme to describe the mobile portion of the system and Thyme-Infra as the infrastructure side, in order to facilitate the distinction between those parties.

Although we are making use of infrastructure services, it is not our intention to overwhelm them with data storage requests nor offloading the entirety of data synchronization process to them, effectively transforming those nodes in cloud-like devices. As stated throughout the rest of this chapter, our goal is to utilize the available wired links between local basic service sets to provide the least amount of metadata and data in order to serve subscriptions from nodes in distinct areas with relevant information.

4.2 System Model

Adding to the Thyme system model [17], we consider a classical asynchronous model comprised of $\Pi = \{n_1, \ldots, n_k\}$ portable nodes, with no mobility restrictions, other than those imposed by the venue they are in, and the natural speed limits of humans. We do not assume any specific radio technology. Nodes communicate by exchanging messages through a wireless medium (e.g., Bluetooth, Wi-Fi ad-hoc, Wi-Fi Direct), and have no access to any form of shared memory. Nonetheless, nodes should be able to establish communication channels with (all) their neighbors. We also consider the classical crash-stop failure model, in which nodes can fail by crashing but do not behave maliciously.

Likewise, the system will incorporate $\gamma = \{i_1, \ldots, i_m\}$ infrastructure nodes, ideally connected through wired means among themselves, wherein each one is responsible for managing, overlapping or disjoint, basic service sets in different areas. These nodes are also able to communicate with cloud services or external databases for long-term persistent storage or analytics purposes, if desired by the infrastructure engineer.

Published object data is considered immutable. Due to the unreliable nature of wireless communication mediums, our system notifies subscribers of all relevant published data as completely and faithfully as possible, i.e., missing some notifications is permitted because applications are not expected to be mission-critical. We also assume nodes' clocks to be synchronized (with a negligible skew). At the same time, mobile nodes are able to publish data to/subscribe to items from other areas of the venue, by communicating with the infrastructure nodes in order to reach distant BSS's.

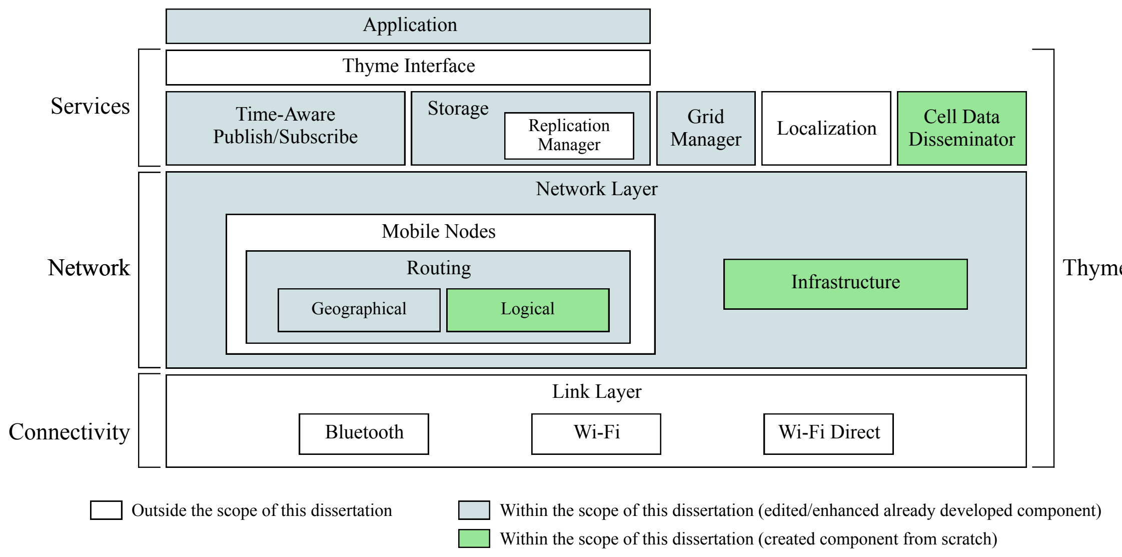


Figure 4.1: Thyme's Updated Architecture. Adapted from [18].

4.3 System Architecture

4.3.1 Thyme

In this section, we present and describe the augmented Thyme architectural structure (Fig. 4.1), based on the authors' proposal in [18], focusing on the layers that were recently created or enhanced during the context of this thesis. The current iteration of Thyme contains the following layers:

Application. Thyme was developed to be a mobile application-agnostic framework with the goal of offering a generic service at the Edge to power a variety of use cases. Therefore, Thyme can be directly imported into any kind of Android 5.0+ application's software stack.

In [18], the authors created a photo sharing application to showcase not only what the framework can do but also as a way of testing Thyme's performance in a real-world scenario (Section 5.1). In this dissertation, we performed some slight changes to the backend of application itself, such as its module dependencies and configurations file, in order to accommodate for the addition of Thyme-Infra nodes and its interactions.

Time-Aware Publish/Subscribe. This module provides the Time-Aware Publish/Subscribe interface to the application in order for the user to be able to persistently share content between devices. As defined in [18], through this P/S layer it is possible to specify a time interval in which subscriptions will be active, whether that interval is in the past, present or future. Furthermore, this layer is responsible for managing the client's subscription and publish requests as well as all the notifications associated with them.

Within the context of this dissertation, this architectural layer suffered an enormous amount of changes, enhancements, bug fixes and additions in order to also take into account the utilization of resources from the infrastructure layer during the execution of the P/S operations.

Storage. Thyme fuses the Publish/Subscribe infrastructure with the actual storage system by using the virtual nodes to store published content and subscriptions, while cells act as virtual P/S brokers. Although minimally, this layer was updated throughout the course of this thesis in order to allow the transmission of download operations from the mobile nodes to the infrastructure, and vice-versa. These processes will be outlined in greater detail in Sections 4.8.2 and 4.9.7.1, respectively.

Grid Manager. This module is responsible for dividing and managing the geographical space divided by cells, which act as virtual nodes representing every physical node that resides within the same geographical space. However, with our proposal of "Logical Cells" (Section 4.4.1), we no longer define cells as a strict geographical boundary in which nodes must lie within to be considered active members of that cell. Thus, this component was heavily updated to comply with the newly introduced paradigm.

Cell Data Disseminator. This newly introduced component is responsible for keeping track of the subscriptions and downloads arriving at the device's associated cell and periodically broadcasting the subscription needs and popularity information to the nearest infrastructure node. Furthermore, the Cell Data Disseminator will also execute a coordination scheme with the rest of the cell nodes in order to elect only one of them to effectively disseminate this knowledge to the Thyme-Infra instance, avoiding the duplication of transmissions. This component, replication scheme and associated data will be described in Section 4.5.

Network Layer. This super-layer was created to support the asynchronous communication between Thyme instances in order to establish the P/S storage system as well as the discovery process of neighboring nodes through the broadcast of *hello* messages. Initially, in the first iteration of Thyme [18], this layer was simply composed of the mobile nodes networking layer for peer-to-peer interactions. However, as the goal of this dissertation is to augment the mobile framework by making use of previously deployed infrastructure resources, there was a need to update this layer and create the respective sublayers for the mobile and infrastructure interactions.

Mobile Nodes Network Layer. As previously described, the Mobile Nodes Network layer allows the communication between mobile devices running Thyme in their front-most application. This component was proposed [35] and fully developed by a third-party group associated with our research project, *"Hyrax: Crowd-Sourcing mobile devices* *to develop edge cloud."* [1], and was imported into Thyme to act as its communication foundation. Furthermore, this layer also offers a custom node identification service that uses logical addresses, instead of more classical approaches such as a TCP/IP address for each device. This topic will be discussed in Section 4.9.5.

Nonetheless, this component did not receive any updates since the last iteration of Thyme.

Routing. The Routing layer was developed with the goal of managing the transmission of messages from one point of the system to another, by choosing the most appropriate dissemination route. In the previous version of Thyme, this layer was simply composed of a Geographical Routing component but, as what happened to the Grid Manager layer, the introduction of the "Logic Cells" paradigm led to the segmentation of this layer into two distinct modes: the previously created Geographical Routing and the recently implemented Logical Routing. Furthermore, in order to use this module within any of the components of the system, independently of the mode it is currently running on, we fundamentally changed the structure of this service by creating an abstract component, *Routing*, in which *GeographicalRouting* and *LogicalRouting* both descend from in order to make its utilization as transparent as possible to all other system components.

- ***Geographical Routing***: As described in Chapter 3, the network's physical space is divided into a grid and all nodes within the geographic boundaries of a cell collaboratively act as a virtual node. Messages are addressed to geographic locations, thus routed to the cell that contains the location of the message destination through a multi-hopping routing algorithm.

- ***Logical Routing***: On the other hand, the Logical Routing subcomponent no longer needs to create a multi-hopping dissemination strategy since cells are no longer defined by a geographical "fence" and each node can contribute to any of the available cells independently of its physical location. Therefore, by using the infrastructure node's Wi-Fi communication means, it is possible to send a message from one node directly to another, without any intermediate routing.

Infrastructure Network Layer. This component, created especially for the latest version of Thyme, allows the communication with the nearest infrastructure node running an instance of Thyme-Infra, if present. By utilizing the resources from the infrastructure, the Publish/Subscribe paradigm implemented by Thyme needed to be augmented to also take into consideration this new type of resources. Thus, components like the T-P/S heavily rely on this network layer to transmit its P/S operations to the infrastructure.

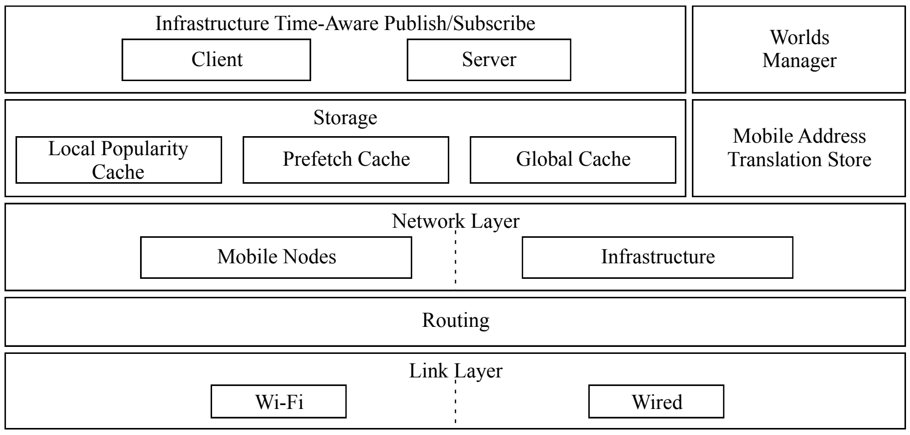


Figure 4.2: Thyme-Infrastructure Architecture.

4.3.2 Thyme-Infrastructure

On the other hand, the proposed framework (Fig. 4.2), Thyme-Infra, to be used on top of previously deployed infrastructure Base Stations is composed of the following layers:

Infrastructure Time-Aware Publish/Subscribe. This layer is the materialization of our proposal to bring Thyme's Time-Aware Publish/Subscribe component to the infrastructure, by leveraging the computational, storage and energetic capacity of these type of resources for this new P/S which are much more powerful when compared to the mobile nodes. At its core, the Infrastructure T-P/S layer will process regular subscription, download and unpublish operations from its clients, just like Thyme would (workflows described in Section 4.8). However, our novel implementation for the Publish/Subscribe paradigm at the infrastructure uses a proactive approach to eventually serve relevant items to the clients' subscriptions by probing the entire infrastructure layer: the Thyme-Infra node will keep track of what its clients are currently subscribing to and periodically disseminate this trend to all other infrastructure nodes with the goal of receiving items that are considered popular within their users, but still relevant to the current node's local users interests, in the near future. By doing so, we augment the amount of information any user can access to by expand a subscription's range to the entire system. We call this entire process "Global Publish/Subscribe" and will be detailed in Section 4.6.

The functionalities of the Infrastructure Time-Aware Publish/Subscribe component are logically divided into two distinct pieces: the client and the server. The Infra T-P/S Client module is responsible for receiving the subscription needs from the mobile nodes connected to the current infrastructure node and broadcasting those needs to all other infrastructure nodes in the system. Contrarily, the Server module is responsible for replying to regular subscription requests from local nodes as well as download operations from both local clients as well as other nodes from the infrastructure. Furthermore, in order to serve those requests, this component is also responsible for deciding which local items are going to be replicated in its local storage. Finally, the Server will process incoming subscription needs requests, originated from other infrastructure nodes' respective T-P/S Client layers, and reply to them with relevant local popular items to be consumed.

Worlds Manager. Unlike Thyme's core world discovery process, which involves users communicating with their peers to know which worlds are currently running, in Thyme-Infra mobile nodes need to inquire with the nearest infrastructure node to obtain the list of available Thyme worlds. This layer is responsible for maintaining a global view of the system, in terms of the active worlds, as well as serving incoming "world probe" messages from the users. Furthermore, this component will need to notify every other Thyme-Infra node when one of its local clients decide to create a world that will be available system-wide. This layer and its workflow will be described in depth in Section 4.4.2.

Storage. The Storage service intent is to maintain and provide an interface to the multiple Thyme-Infra underlying Publish/Subscribe caches, used for different stages/interactions, to all of the system's components, whether to perform simple lookups or to edit their contents altogether. Since these caches will be used to augment and support Thyme's T-P/S paradigm, as well as their associated data types, in the infrastructure, each of them was implemented to be capable of storing a predetermined amount of both metadata and data entries. The Storage layer contains the following caches:

- ***Local Popularity Cache***: Cache containing the most popular items within the *AP's BSS* for easier access, allowing the local clients to obtain those objects quickly and directly from the infrastructure instead of having to inquire their neighbors for a replica. This cache will also be used to serve the subscription needs from remote clients (Section 4.6, Step 1 and 2);

- ***Prefetch Cache***: This cache has the purpose of storing data and metadata entries from objects originated from remote clients. Since this cache will periodically and constantly receive data from the rest of the system, its goal is to allow the current Thyme-Infra node's local clients to discover and obtain items from other parts of the system (Section 4.6, Step 3);

- ***Global Cache***: Cache used to store entries that were initially inside the *Prefetch Cache* and were later considered relevant (i.e. specifically downloaded) by at least one of the infrastructure node's mobile clients. As described, the contents of the *Prefetch Cache* are expected to have a considerable turnover rate. Thus, moving objects from the former cache, that were deemed relevant, into the *Global Cache* guarantees that the specified data is stored in a more persistent way for later access and discovery from other users (Section 4.6, Step 3).

Moreover, the inner logical structure of the caches themselves will be outlined in Section 4.7.

Mobile Address Translation Store. As previously stated, Thyme's Mobile Nodes Network layer utilizes a custom address definition which is used to identify each of the participating mobile nodes, instead of a more classical approach such as a TCP/IP address. However, that particular networking layer was developed in the context of the *Hyrax* research project [1] with only mobile devices in mind, therefore we were not able to utilize it within Thyme-Infra which meant using a distinct approach for the communication layer. Thus, the Mobile Address Translation Store has the single purpose of translating those logical addresses into physical ones that the infrastructure is able to comprehend and use when trying to establish a communication channel with a mobile node. This component will be described in greater detail in Section 4.9.5.

Network Layer. Just like Thyme, Thyme-Infra's Network layer is divided into two parts, the Mobile Nodes' and the Infrastructure's, and has the goal of allowing not only the infrastructure nodes to communicate wirelessly with its mobile clients and through wired links with other base stations but to also process incoming messages from those sources. In order to achieve a system that is able to process multiple requests concurrently, each Network layer's subcomponents were developed with multithreading in mind. Furthermore, the infrastructure developer is able to specify the concurrency level of the Mobile Nodes and Infrastructure layer independently, in order to best fit to its system's needs (Section 4.9.2).

This component was developed to be a general-purpose networking layer with the goal of receiving and transmitting messages of any kind and format, avoiding hard-coded checks and parsing mechanisms, by having each system component register a *listener*, for every type of message that it is interested in receiving, which will be called when a message matching the listener's associated identifier arrives at the infrastructure. Thus, when a message is received by Thyme-Infra, the Network layer starts by reading all the data, parses the first header bytes which indicate the message's type and then, using this information, searches all active listeners for that type and calls them with the raw message bytes and the identifier as parameters. Therefore, the parsing and repackaging procedure to convert the raw bytes into a concrete message object is delegated to the respective component that registered the *listener*.

Routing. Depending on which of the components that call it, whether it is Mobile Nodes Network Layer or the Infrastructure Network layer, the Routing layer will execute custom algorithms to choose the best way to communicate with a specific mobile node — whether directly or indirectly — or a cell as whole (processes described in greater detail in Section 4.9.7), and wired routing mechanisms to propagate infrastructure packets to the correct access point, respectively.

Link Layer. The layer responsible for receiving the messages generated by the Network, and subsequently the Routing Layer, converting them into the correct packet definition

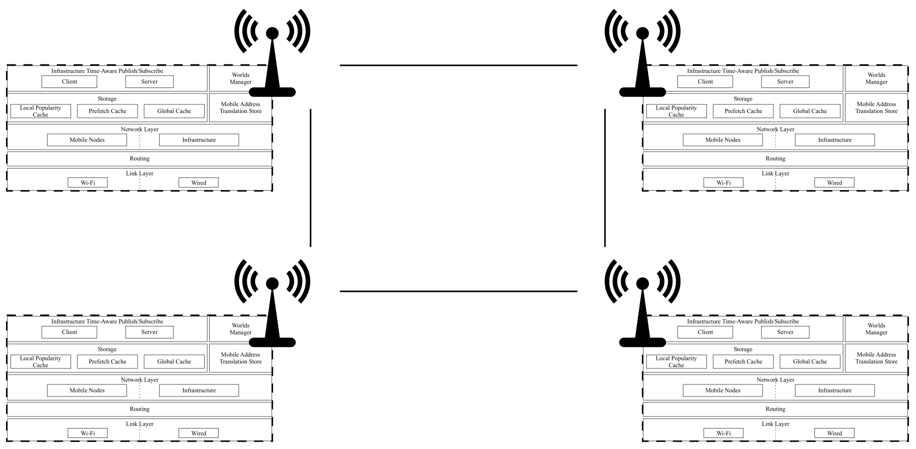


Figure 4.3: A symmetrical system where each infrastructure node runs its own independent Thyme-Infra instance.

depending on the requested type of communication protocol, such as Wi-Fi and wired, and flushing/forwarding to the specified destination. When communicating with other infrastructure nodes it will use the wired communication protocol, while using the Wi-Fi protocol when transmitting messages to the mobile clients.

Finally, just like the Thyme scheme, each infrastructure node will run its own independent Thyme-Infra instance, without any form of direct access to a shared storage or computational unit, or even a central coordination system (Fig. 4.3).

4.3.3 Interaction Between Thyme-Infra Components

As seen in Fig. 4.4, there is an entire web of interconnections between the Thyme-Infra layers that allows the whole system to work in coordination.

Starting with the core component of the Thyme-Infra system, the *Time-Aware Publish/Subscribe* layer is responsible for receiving and processing subscription needs, subscriptions, downloads and unpublish messages from the users as well as messages from other Thyme-Infra nodes, which leads to a direct dependency of the *Network* to perform all of this I/O operations.

Regarding *Time-Aware Publish/Subscribe*'s Client sublayer, the utilization of the *Storage's Local Popularity Cache* is directly tied with its dissemination of its clients subscription needs trend to all other infrastructure nodes. It also communicates with the Server counterpart to pass on other relevant information sent from its mobile users. On the other hand, the Server submodule also interacts with the *Storage* layer in a more complete way, when compared to the module's Client: stores, looks up and updates its P/S caches based on the clients' needs during multiple stages of the system, effectively maintaining and managing the *Storage* component.

As described previously, the Thyme-Infra *Network* layer was created to be a general-purpose component and therefore does not contribute to the maintenance and population

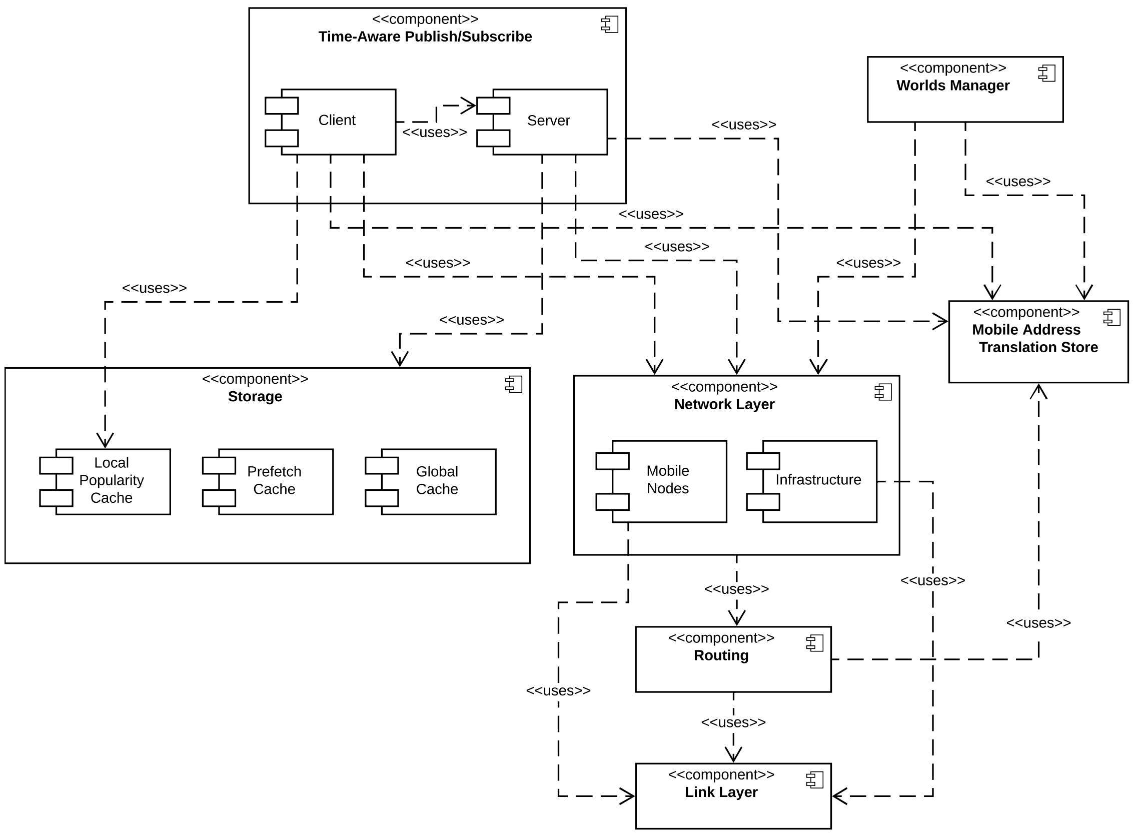


Figure 4.4: Interactions between the Thyme-Infrastructure modules.

of the *Mobile Address Translation Store* since it does not know how to parse incoming messages and extract the logical address information relevant to the translation store, thus the population of the latter component needs to be delegated to other components that know how to perform those activities. Since both the Client and Server submodules of the *T-P/S* layer are expected to receive a considerable amount of messages from the mobile nodes throughout the execution of the system, both of these parties will be re-sposible for communicate directly with the *Mobile Address Translation Store* to store the messages' origin logical address and the associated physical address.

In order to receive "world probe" messages from the local mobile clients and reply to them with the list of active world in the system, the *Worlds Manager* component directly uses of the *Network* layer to execute this request/reply workflow. In addition, just like the *T-P/S* layer, this layer also communicates with the *Mobile Address Translation Store* whenever it receives a message from a mobile node with the goal of contributing to the address store by caching the node's logical and physical address.

The *Network* component is composed of two distinct submodules, *Mobile Nodes Network* and *Infrastructure Network*, and depending on the communication workflow, transmission initiated by the infrastructure itself or a reply to an incoming message, this layer will interact with the *Routing* module in order to perform the appropriate routing mechanism or directly communicate with the *Link* layer to flush the reply to the already open communication channel, respectively.

Consequently, the *Routing* module will outline the appropriate routing mechanism depending on which of the *Network* submodule that called it. Therefore, if the *Infrastructure Network* module interacted with the *Routing* layer, the latter will compute which is the correct infrastructure node destination address and pass this knowledge on to the *Link* layer to transmit the packet. Alternatively, if the *Mobile Nodes Network* sublayer was the one that initiated the routing interaction, this module will lookup the contents of the *Mobile Address Translation Store*, that were populated by all the other components that use it, to be able to translate the logical address specified by the *Network*'s destination address into a physical one that the bottom layer, namely the *Routing* layer, is able to comprehend and process the transmission.

4.4 A Revision of Thyme's Core Features

With the introduction of a whole new set of powerful resources, namely infrastructure nodes, there was a need to implement some changes to the Thyme specification in order to accommodate to the new paradigm. Among all alterations made to the Thyme's core functionality, the most radical ones where: transforming the geographical-based cells grid to a logical layout, the creation and discovery of worlds and the enhancement to the MetaData object used throughout the entire system.

4.4.1 Logical Cells

A fundamental property of Thyme is the fact that it makes use of a geographical division system, in the form of a grid, composed of multiple disjoint and distinct cells. In order to achieve and manage this distribution, Thyme uses the concept of a *Geographic Hash Table*. For the context of the system this approach is extremely advantageous, considering the range limitations of peer-to-peer communication means, since the guarantee that the nodes within the same cell are relatively close to each other ensures, with a certain degree of confidence, the successful communication between these parties in order to allow a correct and persistent maintenance of each cell's indexed data. For requests addressed outside the scope of the current node's cell, Thyme uses the *GHT* to calculate a multi-hopping strategy from the origin to the destination cell.

However, this approach becomes inappropriate with the introduction of the infrastructure layer, which in turn, provides the Wi-Fi interface to its mobile nodes. Thus, independently of the device location within the access point's coverage area, the communication cost of point-to-point interactions using this technology will tend to a negligible value. Consequently, the previously employed geographical grid, and the respective geographic routing algorithms and mechanisms, was removed and the concept of a Logical Grid and corresponding *Logic Hash Table (LHT)* were introduced: the cells, and their nodes, are no longer defined by strict geographic borders but through a *UUID* [36] that

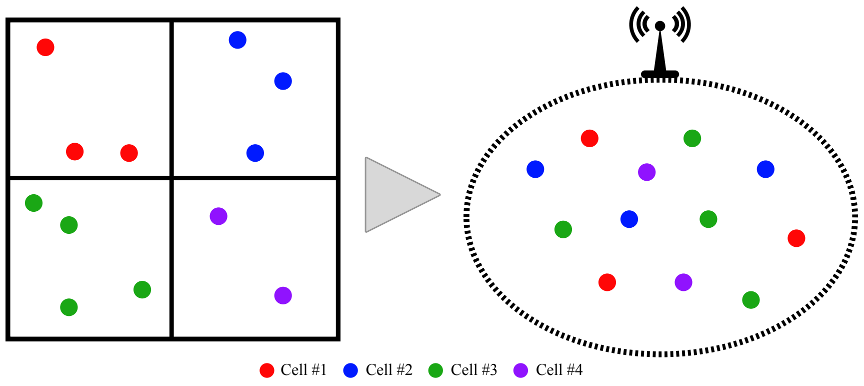


Figure 4.5: Conversion from a Geographic-based cells layout of Thyme to the newly introduced Logical cells in Thyme-Infra.

is generated per device during its startup which will be used for each node to know and inform its neighbors the cell, calculated via that unique identifier, which it belongs to (Fig. 4.5).

With the introduction of this new communication protocol and partitioning logic, one of the biggest preoccupations of Thyme, i.e. mobility-awareness and mitigation (Section 3.5), becomes essentially irrelevant within an *AP's BSS*: nodes can freely move within the range of the infrastructure node as there won't be any housekeeping or mobility notifications involved since cells are logically-defined. Nonetheless, churn continues to be an ongoing threat to the health of the system, reason why we will still employ the replication mechanisms (Section 3.3) in our proposed updated solution.

4.4.2 Creation and Discovery of Global Worlds

Considering the fact that we are trying to join multiple groups of users in different locations, within the same venue, it no longer makes sense to confine the world search process, as described in Section 3.6, to only the nearest clients. This is due to the fact that there is a possibility that a world was already created in another *BSS* which the current client might be interested in joining, even though no other neighbor is currently participating in it. With this in mind, before the new user is ready to join the system, it will start its journey by sending a *World Probe* message through *Multicast* to the nearest infrastructure. Since the clients do not initially have knowledge of which infrastructure node is closest to them, when the infrastructure node boots up it will start to listen indefinitely to messages in a predefined *Multicast* group specified by the framework itself (of which the clients also know the IP and port of this group beforehand). In order to avoid network congestion as a result of the routing process of the *Multicast* messages sent by the mobile nodes, the messages will only have 1 *TTL* as we are expecting the infrastructure node, if present/available, to be the one offering the Wi-Fi service or another device directly connected to the access point.

When the infrastructure node receives the *probe* message, it will then reply to the client with the global list of all worlds created within the entire system, in the format $\langle id_{\text{world}}, name, nr_{\text{cells}} \rangle$. Consequently, when the user receives this information it will have the option to directly join one of the received worlds or, if none is considered relevant by the client, create a new one. In the event that two, or more, infrastructure nodes are within one *TTL* away from the user, the client will consider the one that replies first with the worlds data to be the nearest. Furthermore, since this is the first message received by the infrastructure itself, the client will also cache the infrastructure node's physical address (i.e. IP and port), from the incoming packet, for future communications and, from this point on, will interact directly with it without the need to use multicast dissemination.

To create a new world that will be available to all current and future users, in any location of the venue, the client will send a *Global World Creation* message to the nearest infrastructure, with the corresponding world name. Then, the *AP* will generate a unique ID in order to avoid conflicts with the already available worlds and, in parallel, will confirm the creation of the world to the user and notify all other infrastructure nodes in order for them to locally mark the existence of this world.

As explained, the infrastructure node is the one that specifies and returns the number of cells of each world (nr_{cells}), instead of the client during the creation process (as employed by the previous version of THYME). With this addition, the infrastructure node can modify on-the-fly the number of cells by increasing it when the estimated number of clients connected is high or decrease when this number is low, with the goal of maintaining the system performance and per-node and cell usage, independently of the total number of users.

4.4.3 Augmenting the MetaData Object

The MetaData object is one of the most important data types in THYME. It acts as a small preview of a given file that the client always needs to obtain first and only then it is able to request the associated file bytes. The usage of something like this is ideal for the system's P2P Publish/Subscribe system since it contains a small amount of primitive fields (as specified in Section 3.2), thus MetaData objects can be transferred freely around the system without degrading the overall performance, seeing that they are expected to have much smaller sizes than actual data items, such as pictures or movies.

However, with the addition of infrastructure nodes and the ubiquitous nature of the MetaData itself within the system, this data type required small enhancements during the development of THYME-INFRA. Among all added fields, the most important one is the *sourceInfrastructure* which holds the nearest infrastructure physical address where the specified item was created. The relevance of this change is twofold:

1. As the number of infrastructure nodes grows in the system, it is expected that the number of users connected to them will also increase and, consequently, the number of published items will potentially follow this trend. Therefore, as files flow from one *BSS* to another, the probability of an hash collision for two different objects

is significantly higher. Thus, we will include the *sourceInfrastructure* field while calculating an object's hash in order to avoid such complications;

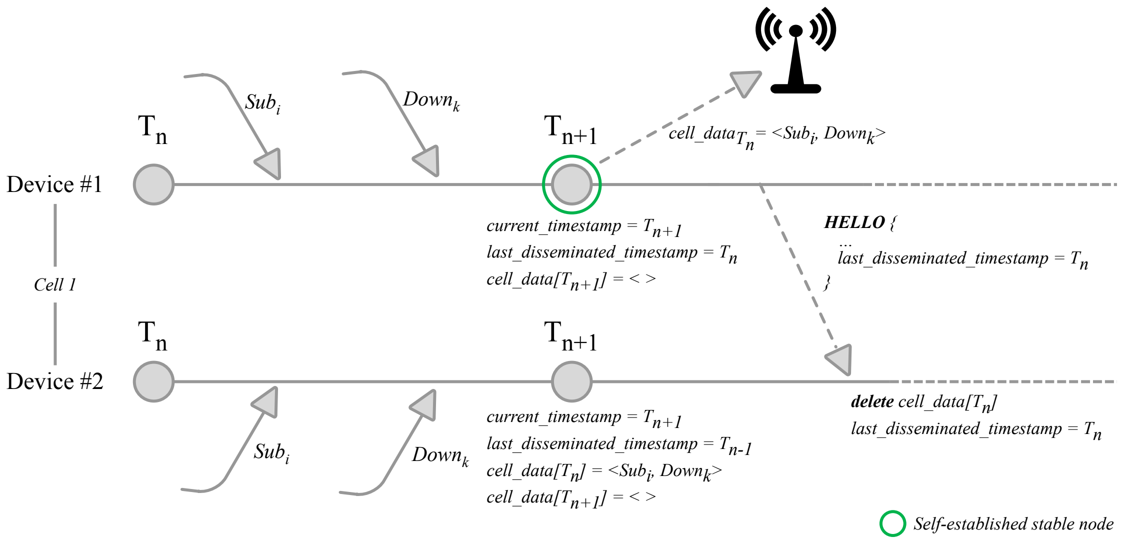
2. On the other hand, by appending the nearest infrastructure address to an item's MetaData object it will be possibly for any *AP* to route a download request to the correct infrastructure node.

4.5 Estimation and Dissemination of Local Subscriptions and File Popularity

Each cell in the *LHT* will maintain and periodically batch-disseminate its *cell data*, which is composed of the *popularity index* (number of downloads, by object, whose data is indexed by the current cell) and the *subscription needs* (total number of subscriptions received by *tag*), to the nearest infrastructure node.

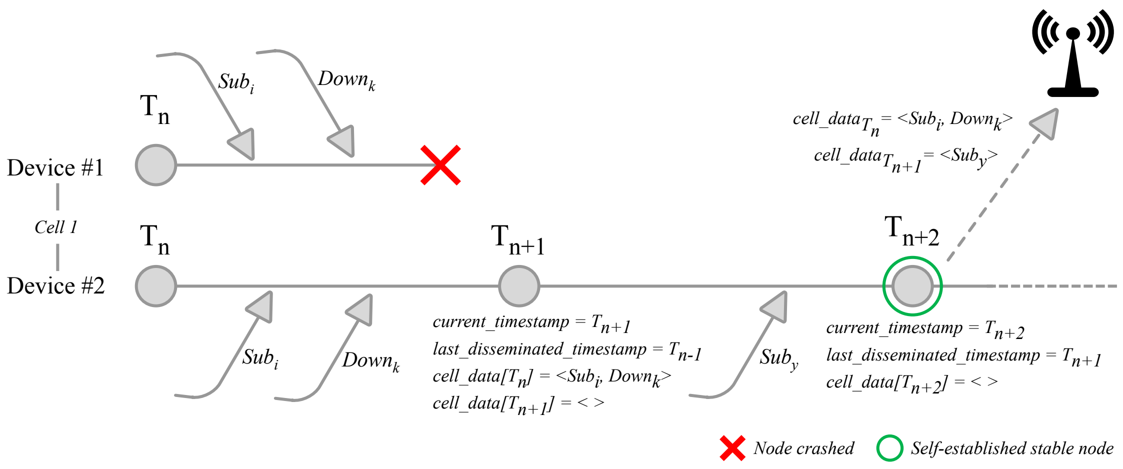
In order to determine a single cell node that will be responsible for disseminating these informations to the *AP* without any kind of strict and computationally-heavy coordination mechanism within all nodes of the same cell, we will make use of something that we internally call a *stability index*. Each device will locally store hardware information from itself and its neighbors (properties such as the battery percentage and others [26, 27]) which will indicate the probability of a node leaving the network voluntarily (e.g. through movement) or involuntarily (e.g. through shutdown due to low battery). These informations will then be appended to the *hello* messages that are already periodically sent as part of Thyme's nodes maintenance process. Whenever the *cell data* dissemination timer triggers, each node within the cell will compare its calculated *stability index* with the ones that were received by its neighbors: if one considers to have the highest value it will take the initiative to start the dissemination process. In the event that two or more nodes have the exact same *stability index* during the comparison check, the one with the lowest identifier will win.

With the goal of implementing a fault-tolerant mechanism to prevent unwanted behavior when nodes that would have been established as the stablest would crash, before the other neighbors noticing the failure, we will employ an approach based on *timestamps*. Thereby, each *cell data* transmission cycle will have assigned a unique timestamp which will also be transmitted to the infrastructure node along with all the *cell data*. Then, after a successful transmission to the infrastructure, the self-established stable node will append the *timestamp* that was just successfully disseminated to its *hello* messages, to act as a confirmation, for the other cell nodes to delete their local data with confidence upon arrival, since they were already transmitted (Fig. 4.6). Even if the to-be established node for the current cycle terminates unexpectedly, its neighbors will have enough time to acknowledge this situation and, therefore in the next dissemination cycle, a new node will be designated and will disseminate not only the *cell data* of the current *timestamp* but all other *cell data*'s from *timestamps* that did not receive any confirmation (Fig. 4.7).



Device #1
Cell 1
Device #2
T_n
T_{n+1}
Sub_i
$Down_k$
$cell\_data_{T_n} = <Sub_i, Down_k>$
current_timestamp = T_{n+1}
last_disseminated_timestamp = T_n
cell_data[T_{n+1}] = < >
HELLO {
...
last_disseminated_timestamp = T_n
}
current_timestamp = T_{n+1}
last_disseminated_timestamp = T_{n-1}
cell_data[T_n] = <Sub_i, $Down_k$>
cell_data[T_{n+1}] = < >
delete cell_data[T_n]
last_disseminated_timestamp = T_n
Self-established stable node

Figure 4.6: Example sequence of the stable node determination and dissemination process of the cell data.



Device #1
Cell 1
Device #2
T_n
T_{n+1}
T_{n+2}
Sub_i
$Down_k$
Sub_y
$cell\_data_{T_n} = <Sub_i, Down_k>$
$cell\_data_{T_{n+1}} = <Sub_y>$
current_timestamp = T_{n+1}
last_disseminated_timestamp = T_{n-1}
cell_data[T_n] = <Sub_i, $Down_k$>
cell_data[T_{n+1}] = < >
current_timestamp = T_{n+2}
last_disseminated_timestamp = T_{n+1}
cell_data[T_{n+2}] = < >
Node crashed
Self-established stable node

Figure 4.7: Example sequence of the stable node determination and dissemination process of the cell data when a to-be self-established stable node terminates unexpectedly.

Thus, a node that joins the network will have to wait for a predetermined amount of time with the goal of tracking the *stability index* of all other cell nodes before being able to join the pool of stable-ready nodes and contribute to the dissemination process.

Since the described process has weak and eventual coordination properties, it is possible for more than one node within a cell to be considered and self-established as a stable node. Let's consider the following scenario: device 1 has a *stability index* of 77 and device 2 has 68; both devices send *hello* messages to each other and device 1 considers itself as the stable node at the moment; then, the user of the second device uses a power bank to charge its equipment and, as a consequence of that, increases its *stability index* to 81; in the meantime the *cell data* dissemination timer triggers for both of the devices before a new round of *hello* messages are transmitted with the latest data; consequently, device 1 considers itself as a stable node since it is comparing its 77 value to device's 2 old 68 value, while device 2 also considers itself as a stable node since its current stable value is 81, resulting in both devices potentially sending the exact same *cell data* to the infrastructure. To avoid duplicating and skewing subscription and popularity information in cases like this, the infrastructure node itself will keep track of the latest received *timestamp* for each cell and will ignore *cell data* messages if the associated *timestamp* was or is already being processed.

Furthermore, when the infrastructure node receives *cell data*, the associated subscription needs and popularity data will be stored and incremented locally in two private tables (*localSubscriptionNeeds* and *localPopularityInformation*, respectively). Later on, after receiving a significant amount of popularity information, the infrastructure node will periodically choose the top *N* most popular items, send a download request for the associated files (MetaData and bytes) to the mobile replicas and store them in the *Local Popularity Cache* with the goal of facilitating the access to this items by its clients. Thus, by storing the most popular files directly in the nearest *AP*, the number of download requests to these items processed by the local nodes will drop drastically, resulting in a considerable battery consumption cutback, as they will be served by the infrastructure itself. After caching the popular items in the infrastructure, it will notify the cells that are currently indexing the items' metadatas stating that they are currently stored in the nearest infrastructure.

As a result of the periodic nature of this popular retrieval mechanism by the infrastructure, the cache contents will change and follow the needs and trends of the mobile clients as they evolve.

4.5.1 Cell Stable Nodes

The realization of the Cell Stable nodes is not only used to operate the mobile part of the system, by being in charge of disseminating the *cell data* information to the nearest infrastructure node, but also considered critical to the execution and performance of the infrastructure side of THYME. Consequently, each THYME-INFRA instance will keep track of all the current self-established Cell Stable nodes within its *BSS* and will use this information to its advantage since this type of nodes are expected to be more stable when compared to rest of the clients, specifically due to their longevity within the system as well as their battery levels and capacity: whenever the infrastructure node wishes to disseminate information to an entire cell, without a specific destination address, or wants to communicate directly with one of the clients but does not currently know the node's physical address (issue described in Section 4.9.5) it will prioritize, if possible, communicating directly with the cell's stable node for it to locally broadcast the cell-addressed message to the rest of its neighbors or route the node-addressed message to the right destination, respectively (Section 4.9.7). In doing so, the THYME-INFRA instance is able to reduce the amount of times it communicates with stale nodes or clients that are offline or already left the system which would eventually lead to the transmission of an unnecessary number of messages that would fail and to the execution of a retry workflow in search of an active and capable client, by trading off the over-utilization of a single node's resources within each cell. Nonetheless, since the cells' determination of stable nodes is an ongoing process, stable nodes are expected to periodically swap between themselves as their *stability index* will decrease at a faster rate when compared to other nodes. Thus, in the grander scheme of things, the system will eventually benefit from the resources of a wider range of users' devices as time goes on.

4.6 Global Publish/Subscribe Process

Since we are using infrastructure nodes to join multiple remote groups of clients within the same venue and their data into one cohesive and consistent end-to-end storage network, we need to adapt the Publish/Subscribe system adopted by THYME to have not only local, but also global interactions into consideration. Thus, a *global publish/subscribe* approach will need to be employed on the infrastructure layer.

This P/S implementation is composed of three steps (Fig. 4.8): 1. global dissemination of subscription needs; 2. global provisioning of subscription needs; 3. local dissemination of incoming subscription needs reply's data to the mobile nodes.

Step 1: Global Dissemination of Subscription Needs. With the goal of allowing mobile clients within each *BSS* to receive and access not only objects published locally but also relevant items published by other users in distinct venue locations, each infrastructure node will start by periodically broadcasting its clients' *subscription needs* to all other *APs*, via *SubscriptionNeeds* messages. Each message consists of:

$$\langle T_{\text{subs}}, S_{\text{cache}}, SN_{\text{world}} \rangle$$

where

$$SN_{\text{world}} = map\langle world_{\text{id}}, entry_{\text{SN}} \rangle$$

$$entry_{\text{SN}} = map\langle tag, sub_{\text{count}} \rangle$$

- T_{subs} holds the total amount of subscriptions, across all worlds, executed by the mobile clients within the infrastructure node's *BSS* up until the moment of transmission;

- S_{cache} denotes the infrastructure node's maximum *prefetch cache* size;

- SN_{world} represents the users' subscription needs, by world, which, in its turn, contains the total amount of subscriptions (sub_{count}) for each tag (tag).

Step 2: Global Provisioning of Subscription Needs. Upon receiving a *SubscriptionNeedsMessage* from other infrastructure nodes, the *AP* will cache all of this information locally. Subsequently, each access point will periodically initiate its data dissemination and provisioning process to all other infrastructure nodes based on their previously broadcasted subscription needs. Thus, the current THYME-INFRA node will lookup its *Local Popularity Cache (LPC)* for objects relevant to each *AP's* needs (the specified tags) and send, in batch, all of the data and metadata entries found during this process through *SubscriptionNeedsReplyMessages*. This message's format is defined by:

$$\langle\, map\langle tag, \{pair_{\mathrm{m,d}} \mid pair_{\mathrm{m}} \text{ is metadata and } pair_{\mathrm{d}} \text{ is data}\}\rangle\,\rangle$$

By making use of the *Local Popularity Cache* to provision content based on the infrastructure nodes' subscriptions, we can achieve a much better network utilization and performance since we guarantee that only the most interesting (i.e. popular) data is transmitted through the network. Furthermore, the node's lookup process is directly executed in-memory instead of having to request the items to its mobile clients in an on-demand manner which would have a direct impact on the delay of the entire process.

However, as the maximum size of each nodes' caches are totally configurable (Section 4.9.9) and hardware-dependent, and thus possibly distinct, this can potentially lead to cases where infrastructure node *A* sends a *SubscriptionNeedsReplyMessage* with a lot more entries than destination node *B* can hold in its *Prefetch Cache*, resulting in an over-utilization of the network and unnecessarily large packets being sent when a percentage of that data will ultimately be discard by *B*'s overflowing cache. Therefore, we will capitalize on the received *SubscriptionNeedsMessage*'s S_{cache} field and trim the data to be sent, by removing exceeding entries, in order to meet the destination's cache size demands.

Another optimization process we employ during the provisioning and creation process of *SubscriptionNeedsReplyMessages* is to use the subscription count for each tag, which effectively represents the subscription popularity, or relevance, for each tag and calculate/reserve the appropriate size within the packet to hold the relevant entries found. Thus, more popular subscription tags will have more space in the *SubscriptionNeedsReplyMessage* while the opposite end will have less entries or even none. The formula used to calculate each tag's maximum packet entries is as follows

$$Entries_{\mathrm{tag}} = \min\left(\frac{S_{\mathrm{cache}} \times sub_{\mathrm{count}}}{T_{\mathrm{subs}}}, \#LPC_{\mathrm{tag}}\right)$$

where $\#LPC_{\mathrm{tag}}$ represents the amount of entries considered relevant to the specified *tag* within the infrastructure node's *Local Popularity Cache*.

Hence, if we consider the following scenario where an infrastructure node receives a *SubscriptionNeedsMessage* from another *AP* which notes that its *Prefetch Cache* has a maximum size of 50 ($S_{\mathrm{cache}} = 50$), its clients executed 100 subscriptions so far ($T_{\mathrm{subs}} = 100$) in which 70 of them were for tag *GOAL* ($\langle GOAL, sub_{\mathrm{count}} = 70\rangle$), the number of entries for tag *GOAL* to be returned to that node will be 35 (if we recognize that $\#LPC_{\mathrm{GOAL}} \geq 35$).

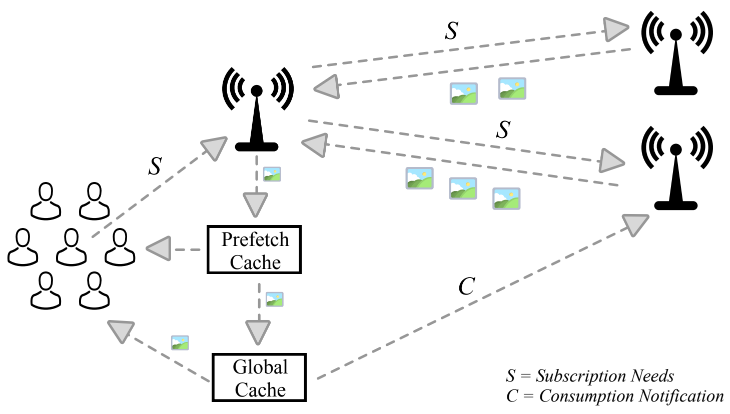


Figure 4.8: Global Publish/Subscribe execution process.

Step 3: Local Dissemination of Incoming Subscription Needs Reply's Data to the Mobile Nodes. When the original Thyme-Infra node receives a *SubscriptionNeedsReplyMessage*, it will start by mapping the received items' tags into the logical cells and send to each one a notification message, namely *InfraNotificationMessage*, with the entries' metadata in order for the cells' nodes to check the ongoing subscriptions and notify the corresponding subscribers of the arrival of the data, depending on their interests. Simultaneously, the received items will also be stored in its *Prefetch Cache* while the local clients get notified and the data will remain there until it receives another *SubscriptionNeedsReplyMessage* that will potentially override the contents of this cache.

Afterwards, if a local client finds one of the received metadata entries relevant and decides to download the associated item's data, by looking at the metadata's "description" field (which acts as a miniature or a preview of the whole item data), it will manifest its intent by sending a download message to the nearest infrastructure node in order to get the bytes and to effectively host a local replica of that data within the current infrastructure's *BSS*. Upon receiving this download request, which means that an object originated in another area was consumed locally, the *AP* will remove that entry from the *Prefetch Cache* and move it to the *Global Cache*. As the turnover rate of the contents of the former cache is expected to be quite high due to the intrinsic nature of the Global Publish/Subscribe Process, which will be even more accentuated when the total number of infrastructure nodes in the system increases, moving the consumed item to the *Global Cache* allows it to be stored in a more persistent manner and be easily accessed by other clients. Further requests to the same item from other clients in the access point's *BSS* will result in obtaining the object directly from this cache. Additionally, since only the items considered relevant by the local users are moved from one cache to the other, there is a high probability for the item to also be considered relevant by other clients since it is expected for geographically-adjacent users to share similar interests to some degree.

Finally, when an entry is successfully inserted into the *Global Cache* as a result of the previous process, the current Thyme-Infra node will notify the item's original infrastructure node that it was consumed by its clients. By doing this, the original *AP* will save this information and will not send the specified item again in the next *n* (configurable value - Section 4.9.9) “Global Provisioning of Subscription Needs” dissemination cycles, thus saving network bandwidth since the item is already in that infrastructure node’s cache.

4.6.1 Infrastructure Non-Overlap Dissemination Contract

As explained in the previous section, each infrastructure executes the “Global Provisioning of Subscription Needs” step which represents the transmission of the *AP’s* data to all other Thyme-Infra nodes in the system based on their clients’ needs. However, if we consider a worst-case scenario where all infrastructure nodes but one decide to send their *SubscriptionNeedsReplyMessage* to the remainder node, at the same time, the system network and the latter *AP* will be overwhelmed with the amount of packets, and their considerable size. The expected outcome would be an over-saturation of the *Prefetch Cache*, since multiple data items will compete with each other for cache space, in a short amount of time, and will evict and overwrite each other. Furthermore, mobile clients will be bombarded with the arrival of metadata to be viewed and filtered (i.e. marked as relevant to be further downloaded), which will also result in an unnecessarily substantial battery consumption peak as well as a poor user experience.

To avoid scenarios like this and the complications associated with them, we will employ a non-overlap dissemination contract: during the startup of each *AP* node, each Thyme-Infra instance will compute its unique, non-overlapping and repeatable dissemination time-slots while taking into consideration all other nodes that will be available in the system. Thus, every infrastructure node will be able to transmit their *SubscriptionNeedsReplyMessages* knowing that, with a certain degree of confidence, no other *AP* will thwart their process and that the destination Thyme-Infra nodes, and their clients, will receive and process the incoming data smoothly and without disturbance.

Although each dissemination window is totally configurable by the infrastructure developer (Section 4.9.9), the duration of an optimal contract should take the following components into consideration (Fig. 4.9)

$$\Delta_{\text{contract}} \geq \Delta_{\text{i}} + \Delta_{\text{pi}} + \Delta_{\text{tm}} + \Delta_{\text{d}} + \Delta_{\text{dt}} + \Delta_{\text{it}}$$

where:

- Δ_{i} is the average transmission duration of *SubscriptionNeedsReplyMessages* from one infrastructure to another;
- Δ_{pi} represents the time used by the infrastructure node to process the incoming message and choose which mobile clients to notify about the items arrival;
- Δ_{tm} denotes the transmission delay of incoming items to the mobile nodes that are potentially interested in them;

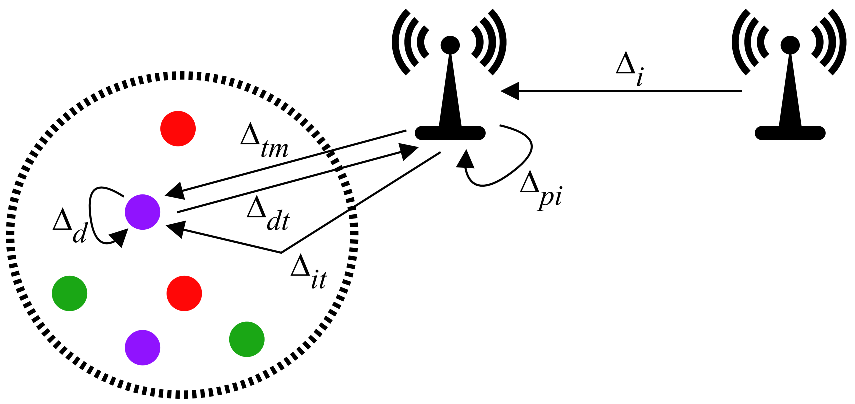


Figure 4.9: Optimal Non-Overlap Dissemination Contract Breakdown.

- Δ_d represents the mobile clients' average decision time to choose whether to download the received item(s) or not;
- Δ_dt describes the transmission duration of the user's decision to download the item to the nearest infrastructure;
- Δ_it delineates the requested item's transmission latency from the infrastructure to the client.

4.6.2 Bringing Time-Awareness to the Infrastructure

One of the major features of Thyme compared to other edge peer-to-peer storage frameworks is its time-awareness: users are able to specify, on their subscriptions, whether they wish to receive items that were published in the past, data that is currently being created or be notified of future topic occurrences.

During the entirety of the design and planning phase of this thesis, it was one of our main priorities to bring this paradigm into the newly developed processes and algorithms of the infrastructure layer. Thus, time-awareness was implemented throughout Thyme-Infra, specifically in the following areas:

- **Cell Data.** Instead of only tracking the subscriptions and downloads, each logical cell's nodes also tracks the subscriptions' time interval (minimum start time and the maximum end time) and appends that to the *cell data* message, described in Section 4.5, to be disseminated to the nearest infrastructure node by the elected stable node in each transmission cycle;
- **SubscriptionNeedsMessage.** The nearest infrastructure listens to all *cell data* messages and stores the minimum subscription start time and maximum subscription end time across all subscriptions, across all cells. Then, when transmitting the clients' subscription needs to all other infrastructure nodes, the current Thyme-Infra instance also appends those values to the disseminated *SubscriptionNeedsMessage*;

- **SubscriptionNeedsReplyMessage.** Now with each infrastructure time-aware information, during the Global Publish/Subscribe's "Global Provisioning of Subscription Needs" step, the current Thyme-Infra node takes into account the subscription times when looking up its *Local Popularity Cache* for data to supply to the infrastructure nodes based on their needs, filtering the entries that have publish dates outside of the received time intervals.

4.7 Infrastructure Publish/Subscribe Caches

As was introduced in Section 4.2 and referred to in subsequent sections, each Thyme-Infra instance contains three different-purpose caches: the *Local Popularity Cache*, the *Prefetch Cache* and the *Global Cache*.

Due to the inherent paradigm features of Thyme, in most cases, using a simple one dimensional caching approach would not achieve an optimal performance and storage utilization for the system. Thus, we proposed and developed a new caching method called *Adaptive Multipart Caching* that best fits the system needs (Section 4.7.1). Furthermore, we will define the structure of each of the three caches in Section 4.7.2. Finally, we will disclose the enhancement of each cache through additional auxiliary data structures for better single-purpose lookups (Section 4.7.3).

4.7.1 Adaptive Multipart Caching

A World in Thyme can be seen as simple photo galleries or shared folders where a user might want to join a single one of them or both, while they can also be seen in a more complex manner, such as distinct applications' environments with disparate data types, flows and even types of users. Thus, since each infrastructure node will be able to accommodate their service to all connected clients, independently of the world/application they are part of, the Thyme-Infra instance has to provide a storage system that is able to adapt to each world's clients needs and, specifically, its population size.

Initially, we considered using a simple one dimensional cache design approach, where there were no disjoint containers to store the data for each world independently. But in doing that, the cache would not take into account each world's needs and every client would blindly compete with users from other worlds for storage space. Instead, we decided to take another storage approach, namely the *Adaptive Multipart Caching* or *AMC*.

In *AMC* there is no single unidimensional cache, but an array of distinct caches, one for each populated world within the *AP's BSS*. The idea is that each container's data is totally independent from the others and its size is completely flexible. That is, if the overall needs of one world changes and it needs a bigger cache size than what it currently has, due to an increase of the world's population for instance, the *AMC* will adaptively increase its size by decreasing the size of the other containers, if needed, in order to

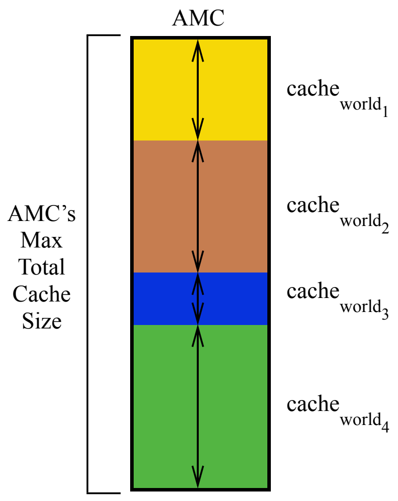


Figure 4.10: Example definition of the *Adaptive Multipart Caching*.

make space for it without ever exceeding the maximum total cache size specified by the developer. The result will be something similar to Fig. 4.10.

When Thyme-Infra triggers the *AMC's* execution, the system recalculates each inner cache's size based on the following formula:

$$AMC_{\mathrm{w,t}} = \frac{(\#entries_{\mathrm{w,t-1}} + \#put_{\mathrm{w}})}{\sum\limits_{ww} \#entries_{\mathrm{ww,t-1}} + \#put_{\mathrm{ww}}} \times S_{\mathrm{cache}} \;\;,\;\; ww \in AMC \tag{4.1}$$

where S_{cache} represents the maximum total cache size, $\#put_{\mathrm{w}}$ holds the number of entries to be inserted in world's w cache and $entries_{\mathrm{w,t-1}}$ indicates the number of entries in world's w cache after the previous execution of *AMC*. Essentially, *AMC* calculates the fraction between the world's to-be expected cache size ($\#entries_{\mathrm{w,t-1}} + \#put_{\mathrm{w}}$) and the total expected cache size ($\sum\limits_{ww} \#entries_{\mathrm{ww,t-1}} + \#put_{\mathrm{ww}}$), if we were to not limit the number of entries being inserted in both cases, and then scales this value to the correct and final world cache size using the maximum cache size (S_{cache}) as the upper bound.

Let's consider the following scenario: an *AMC* cache currently has two world containers, for "gallery" and "myapp" ($ww = \{gallery, myapp\}$), with sizes 150 ($\#entries_{\mathrm{gallery,t}} = 150$) and 50 ($\#entries_{\mathrm{myapp,t}} = 50$), respectively, with 200 acting as the overall cache size limit ($S_{\mathrm{cache}} = 200$). Then, the system decides to batch-insert 100 new items that are now suddenly considered highly popular, 80 of which are from world "myapp" ($\#put_{\mathrm{myapp}} = 80$) and the rest are from the "gallery" ($\#put_{\mathrm{gallery}} = 20$), therefore meaning that the total expected cache size for this particular *AMC*, if we were not to limit by its upper bound after the insertions, would be 300 ($\sum\limits_{ww} \#entries_{\mathrm{ww,t}} + \#put_{\mathrm{ww}} = 300$). Consequently, "gallery"'s new size would be approximately 113 ($\#entries_{\mathrm{gallery,t+1}} = \frac{150+20}{300} \times 200 \approx 113$) while "myapps"'s would slightly increase its storage capacity from 50 entries to 87 ($\#entries_{\mathrm{myapp,t+1}} = \frac{50+80}{300} \times 200 \approx 87$).

As we can see, one of the greatest advantages of using a technique like *AMC*, other

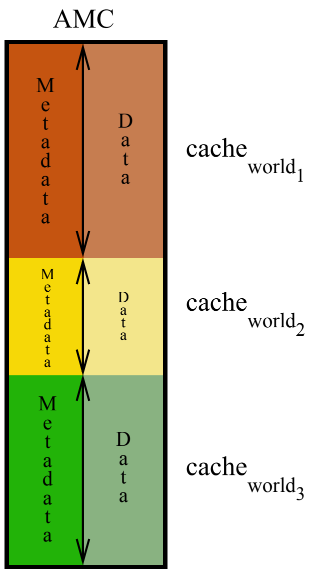


Figure 4.11: Example definition of the *Local Popularity Cache* structure.

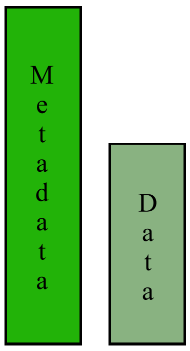


Figure 4.12: Example definition of the *Prefetch Cache* structure.

than the ability to create multiple application-purpose containers within one general cache, is that it dampens the negative impact of scenarios when there is a short burst of a large amount of downloads made from users within a single world on the caches' divisions. Without any additional logic, these scenarios could lead THYME-INFRA to offload a considerable amount of entries to the infrastructure's caches, due to their rising local popularity, resulting in an overshadowing and ultimately an overwrite of other cached data in order to make space for the incoming ones, when this could simply be a non-recurring and a stand-alone event. Thus, with the addition of the *AMC* mechanism, these short bursts will not be able to strip away an high amount of cache space in one sitting, but in a gradual manner. Therefore, if this popular behavior continues, the world's users will be able to gradually increase their world's cache size to fit their needs.

4.7.2 Caches Structural Definition

Local Popularity Cache. The *Local Popularity Cache* is composed of a single *Adaptive Multipart Cache* where each inner caches' entry represents a pair of data and metadata objects, entities which are crucial and ubiquitous to the THYME system (Fig. 4.11). Thus, this *AMC* is defined by:

$$\langle world_{\text{id}}, cache\langle object_{\text{id}}, \{pair_{\text{m,d}} \mid pair_{\text{m}} \text{ is metadata and } pair_{\text{d}} \text{ is data}\}\rangle\rangle$$

Furthermore, in this cache, the *AMC's* partition sizing mechanism is executed at insert-time, during each batch-insert window of the *Local Popularity Cache* (as will be described in Section 4.9.3).

Prefetch Cache. On the other hand, as previously described, the *Prefetch Cache* is used to store the incoming data from each other infrastructure node that replied the current Thyme-Infra instance's subscription needs requests. This is the only cache, out of all three specified in the Thyme-Infrastructure definition, that does not make use of the *Adaptive Multipart Caching* approach. The reasoning behind this design decision is that *AMC* would have no real benefit on this particular cache since the data received, as part of Global Publish/Subscribe's "Global Provisioning of Subscription Needs" step, is already processed and packed by a pseudo-*Adaptive Multipart Caching* algorithm which divides all entries by world, according to each one's needs. Instead, the *Prefetch Cache* contains two inner one-dimensional disjoint caches, one for the metadata objects and another for the data entries (Fig. 4.12). By having distinct caches for the metadata and data counterparts, and allowing the infrastructure developer to fine-tune and give different maximum sizes to each of them, the goal is to increase and encourage the discoverability of new items that are created throughout the entire system since each inner caches evict items independently of each other. This is possible since metadata objects are expected to be considerably lower in size when compared to data entries, thus the metadata cache can store a lot more entities before reaching the same memory/storage footprint of that of the data cache. Consequently, if one data entry is evicted to make space for another, but the associated metadata entry is still cached in the metadata disjoint cache, it is possible for a client to still obtain that data: when asking the infrastructure to return that item, the infrastructure node will verify that the requested data is not stored locally, check the data's metadata information and, based on that, route the download request to the original infrastructure where the item was created.

As there is no explicit separation between worlds' data entries at the cache level, logically, each item in each of the inner caches is indexed by the composite key $\langle world_{\text{id}}, object_{\text{id}} \rangle$.

Global Cache. Regarding the *Global Cache*, its structural definition is similar to the one used in the *Prefetch Cache*, carrying the same discoverability advantages as the latter approach, but each of the disjoint caches — metadata and data — uses the *AMC* partitioning mechanism (Fig. 4.13). Thus, the metadata and data *AMCs* are logically defined by, respectively:

$$\langle world_{\text{id}}, cache\langle object_{\text{id}}, metadata \rangle \rangle$$

$$\langle world_{\text{id}}, cache\langle object_{\text{id}}, data \rangle \rangle$$

Unlike the *Local Popularity Cache* which executes the *AMC* during each batch-insert, entries inserted into the *Global Cache*, as we have explained before, are done so in an individual manner. Consequently, re-executing *AMC* in an item-to-item basis would not be optimal in terms of the performance of the system. Thus, we implemented a new layer to mitigate this issue by only executing the repartitioning workflow after a specific number of puts are executed in a world's cache. This procedure is described in Algorithm 1. In essence, while the entire *AMC* cache is not full, items are directly inserted into the

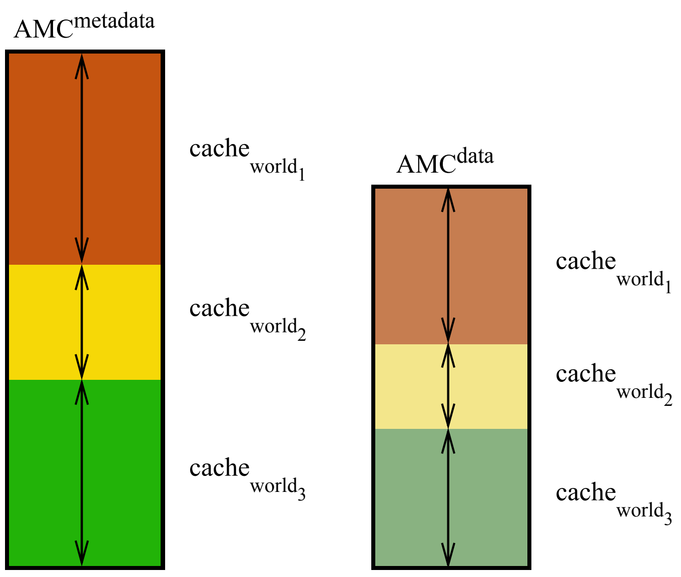


Figure 4.13: Example definition of the *Global Cache* structure.

associated world cache (metadata or data) and its inner size is incremented. When it is full, the item is also inserted in the respective cache, eventually triggering the cache's eviction policy to maintain its maximum size. Simultaneously, the system increments a counter which indicates how many items, for each of the inner world caches, were inserted above its size limit. When this upper bound reaches a predetermined value, specified by $MAX\_PUTS\_ABOVE\_LIMIT$ which is a variable parameter set by the developer (Section 4.9.9), that counter is reset and the *AMC* execution is triggered with the goal of increasing the size of that world cache by the specified value while decreasing the size of the rest of the inner caches to make space for it.

Algorithm 1 Global Cache AMC Workflow

```
1: $putsAboveCacheLimit : \langle world_{id}, integer\rangle \leftarrow \langle\rangle$
2:
3: procedure CACHE(world, item)
4:     if $totalItems() < maxCacheSize()$ then
5:         incrementWorldCacheSize(world)
6:     else if $currentCacheSize(world) \geq maxCacheSize(world)$ then
7:         $putsAboveCacheLimit[world] \leftarrow putsAboveCacheLimit[world] + 1$
8:         if $putsAboveCacheLimit[world] = MAX\_PUTS\_ABOVE\_LIMIT$ then
9:             $putsAboveCacheLimit[world] \leftarrow 0$
10:            $executeAMC(world \rightarrow increase\ by\ MAX\_PUTS\_ABOVE\_LIMIT)$
11:    put(item, getCache(world))
```

4.7.3 Lookup by Tag

Aside from direct object lookups using the object's identifier, there is no built-in way of searching all available metadata entries for a specific world, given a tag, which is essential to support operations like subscriptions (see Section 4.8.1). Thus, each of the three described caches also have an auxiliary data structure specifically created to allow lookups to metadata objects based on a given search tag. This structure is defined by

$$map\langle\langle world_{\text{id}}, tag\rangle, set\{object_{\text{id}}\}\rangle$$

where each $object_{\text{id}}$ can be used to retrieve the real metadata object from the appropriate metadata cache, whether it is an *AMC* or a plain one-dimensional cache. To maintain the consistency of this structure in regards to the metadata entries available in the cache, the maintenance process is twofold:

- **During insertion-time:** every time a new metadata object is inserted into the associated high-level cache, the process loops through all the object's tags and, for every single one of them, inserts the metadata's object identifier into the appropriate $\langle world_{\text{id}}, tag\rangle$'s set;
- **During removal-time:** on the other hand, when a metadata entry is evicted due to a manual or passive (substituted to make space for a new item) removal, the process executes the inverse of the previous operations, which consists of looping through all the metadata tags and removing the associated object identifier's set entry, in the auxiliary structure, within each $\langle world_{\text{id}}, tag\rangle$.

4.8 An (Infra)structured Modification of Thyme's P/S Operations

By introducing the infrastructure layer to Thyme, most of the core Publish/Subscribe operations' received a paradigm shift, in Thyme-Infra, in order to accommodate to these new types of resources. Furthermore, since files can freely move around the entire system and be obtained even if the user is outside the source's *BSS*, operations need to be executed in a local (within an infrastructure node's *BSS*) and global (from one infrastructure node to another) manner. Thus, the following operations received major modifications: subscribe, download and unpublish operations.

4.8.1 Subscription

In addition to sending the subscription message to each cell that is currently indexing the specified subscription tags, as defined by Thyme (Section 3.4), in Thyme-Infra the client will also simultaneously disseminate the same operation to the nearest infrastructure node. However, in the latter case, the infrastructure will process the subscription in a transient manner and will consider it to be a simple lookup operation on the data available within all its caches. Hence, upon receiving a subscription message, the *AP* will obtain every cached item that is relevant to the user, by looking up entries associated with

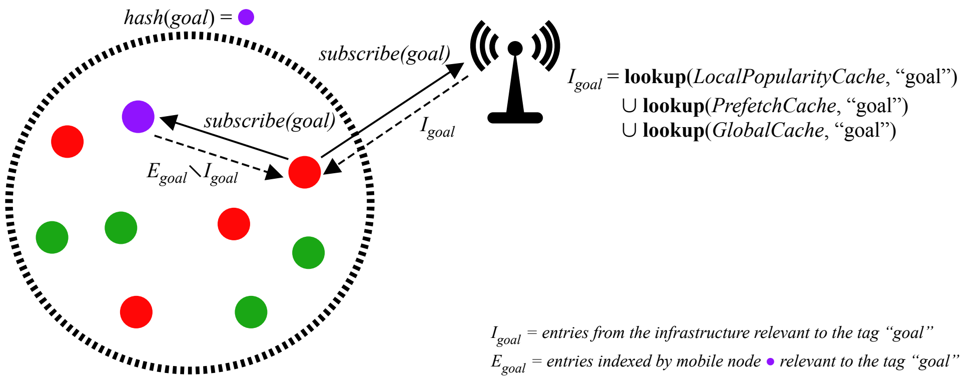

hash(goal) = ●
subscribe(goal)
subscribe(goal)
I_{goal}
$E_{goal} \setminus I_{goal}$
I_{goal} = **lookup**(*LocalPopularityCache*, "goal")
∪ **lookup**(*PrefetchCache*, "goal")
∪ **lookup**(*GlobalCache*, "goal")
I_{goal} = entries from the infrastructure relevant to the tag "goal"
E_{goal} = entries indexed by mobile node ● relevant to the tag "goal"

Figure 4.14: Example of Thyme and Thyme-Infra's subscription workflow when a client executes a subscription operation for the tag "goal".

the specified subscription tags, and reply to client all metadata items found during this process.

However, considering the fact that the same subscribe operation will be transmitted to the mobile nodes as well as the infrastructure layer, there is a high probability of the mobile nodes replying to the subscriber with duplicate entries that were already sent by the infrastructure, or vice-versa. Therefore, in order to avoid the transmission of duplicate data during a subscription process, and the network degradation as a consequence of that, we prioritize the usage of the infrastructure resources, since they can be considered a bit more unrestricted when comparing to the mobile counterpart, with the goal of saving battery consumption of mobile devices. Consequently, the mobile devices will only reply to a subscription message with metadata entries that are currently not available in the nearest infrastructure storage (Section 4.9.6), while the *AP* itself will transmit every item obtained during the described process (Fig. 4.14).

4.8.2 Download

Compared to the previous version of Thyme, the new iteration of this particular operation suffered the most radical transformations due to the increased complexity as a result of the growing number of storage resources and locations by making use of the infrastructure layer. A simplified version of the workflow executed during a download operation can be seen in Fig. 4.15.

In essence, the requesting mobile node starts by inquiring the nearest infrastructure node to check whether the requested item is currently stored in one of its caches: if so, the client obtains the object directly from the infrastructure, otherwise it disseminates the request to the mobile replicas within the same *BSS*. Further, we introduce the distinction between a regular download and a "force download". Let's consider the scenario where a user transmits a download request for an item that was originally published in another *BSS* to its Thyme-Infra node but that object is currently not available there. This would result in the infrastructure not knowing whether the client wants the download

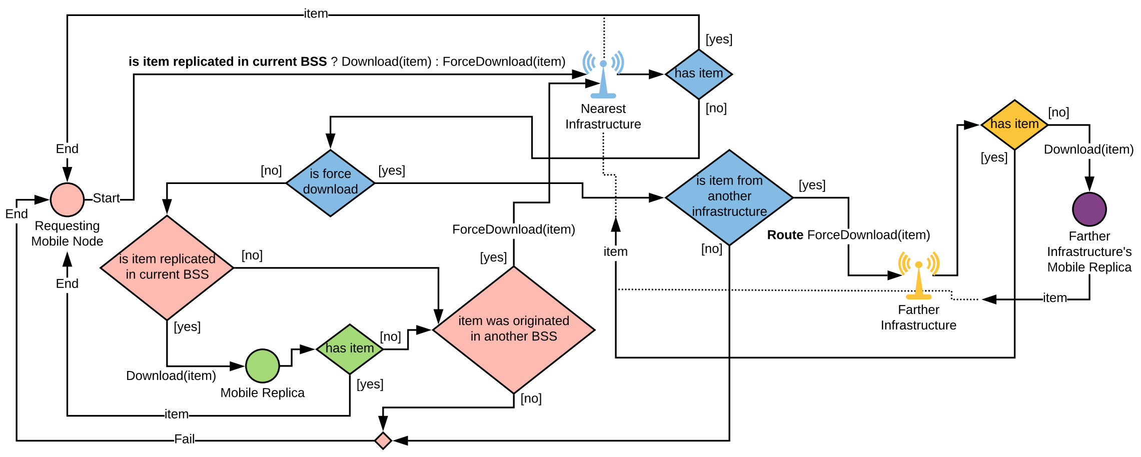


Figure 4.15: Simplified workflow of the download operation triggered by a mobile node (salmon-colored circle). The color in the decision flows indicates which actor is executing them.

to be routed to the infrastructure node where the item was originally created or not: routing the download request to another area if that item is not replicated by any mobile nodes locally would be considered correct, but doing the same thing while having mobile clients with the item stored in their devices would be inefficient and underperforming. Therefore, the download operations sent from the mobile nodes to the infrastructure are defined as follows:

- **Regular Download**: informs the infrastructure to only lookup for the specified item in its local caches and return either the data or a failure notification right away;
- **Force Download**: informs the infrastructure to lookup for the specified item in its local caches but, if the item is not present and was originated in another *BSS* from another infrastructure node, the current infrastructure will route the download request to be processed there.

Additionally, when an infrastructure node receives the reply to a mobile client-initiated *force download* with data from another infrastructure, before routing the requested item to the end-user it will proactively cache the incoming object in the *Global Cache* since it is considered relevant. Thus, consequent requests to the same item, by users in the same *BSS*, will be served directly by the nearest infrastructure node's storage.

Regarding the simplified aspects of the presented diagram, made to streamline the comprehension of the described procedure, when a mobile or an infrastructure node tries to download the item from another mobile replica, instead of asking just one — like the green and the purple circles — as is visible in the image, the actual implementation of this workflow takes into account all the possible local mobile replicas for that item and iteratively asks a subset of them, during the retry procedure, until the data is successfully retrieved.

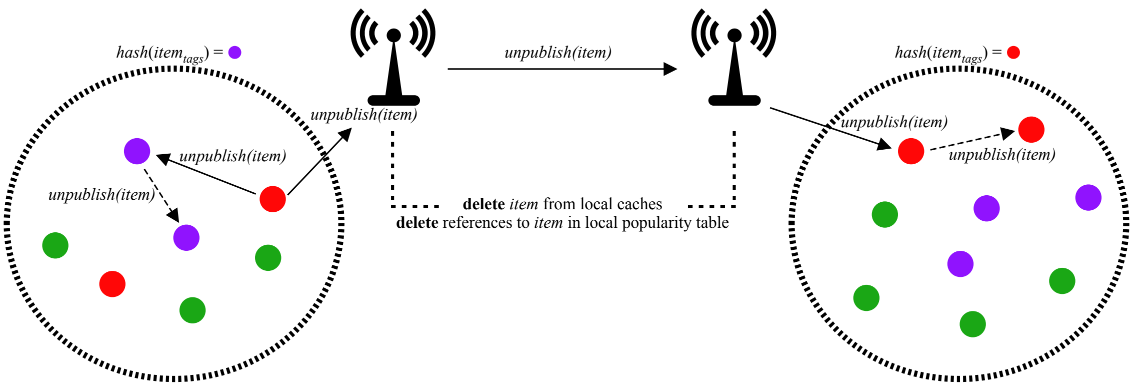


Figure 4.16: Example of Thyme and Thyme-Infra's unpublish workflow, considering that the system is composed of two infrastructure nodes and the item to be unpublished has the tag "goal".

4.8.3 Unpublish

When a mobile node triggers an unpublish operation, the message is transmitted to the item's indexing *LHT* cells and processed normally, according to the specification of Thyme [18]. Nonetheless, since the same file could have been obtained by users in other *BSS*'s (through the process described in Section 4.6), the updated Thyme framework will also, in parallel, notify the nearest infrastructure node in order for the operation to be propagated to the entire system. Thus, whenever another *AP* receives the unpublish message it will start by deleting, if present, the associated entries of the referenced object in any of its caches and propagate the same message to the respective indexing cells for them to execute the operation locally. This workflow (Fig. 4.16) effectively guarantees that an object is indeed unpublished and persistently prohibited from reappearing to any user throughout the lifetime of the system.

4.9 Implementation Details

In this section we will disclose some of the design decisions and implementation details of Thyme-Infra that, while not needed for the overall comprehension of the system, display several engineering aspects and feats involved throughout the development process of this thesis.

4.9.1 Protocol Buffers

In order to facilitate the transition and interconnection between Thyme and Thyme-Infra, as well as the creation of eventual features on both systems, all new and updated messages throughout the development of this thesis were built on top of the Protocol Buffers serialization mechanism, which was already being used by Thyme. Although there is a plethora of serialization mechanism and formats, such as XML, JSON and Java

Serialization, the authors of Thyme note that using Protocol Buffers would best fit their needs [18].

Protocol Buffers [37] is a data serialization mechanism developed by Google, which is platform and programming language-agnostic, enabling a large array of hardware and software solutions to be able to use it interchangeably. As noted by the authors of Thyme during their serialization research, Protocol Buffers has a very strong focus on speed, beating XML by 20 to 100 times faster during transmissions. Furthermore, the packets generated by this protocol are incredibly small when compared to other similar solutions.

Unlike Java Serialization where the class specification also works as the serialization specification [38], Protocol Buffers requires the developer to create an additional *.proto* file which contains a general specification of the object's fields, types and their order within the message. Then, this file needs to be compiled into real serialization code with Google's provided toolchain which, consequently, allows the developer to directly use the generated class and convert its objects into bytes, ready to be transmitted.

4.9.2 Concurrency

Due to the nature of the proposed solution, much like a client-server model which is I/O bound, one of the main goals defined for this thesis was to create a solution that fully utilizes the multithreading architecture of modern hardware to allow the processing of multiple clients' requests at the same time, but also a solution that is concurrently sound. To achieve this, we make use, throughout the entire system, of Java's atomic package [39] classes, such as *AtomicBoolean* [40], *AtomicInteger* [41] and *AtomicLong* [42], to guarantee that specific flags and counters which can be modified by any number of incoming clients' requests are correctly modified and incremented, respectively.

On the other hand, the majority of the data structures employed by Thyme-Infra are from the Java concurrent family [43], like the *ConcurrentHashMap* [44] and *ConcurrentLinkedQueue* [45]. Regarding the *ConcurrentHashMap*, we make extensive use of its *compute* [46], *computeIfAbsent* [47] and *putIfAbsent* [48] methods, with the goal of guaranteeing the atomicity of operations, avoiding scenarios like Listing 4.1, where, even though the data structure and the operations themselves are thread-safe, this logic definition is not, which can lead to incorrect executions flows and outcomes: if two threads execute line 5 concurrently, the outcome will be value 1 in key *test* when the expected outcome would have value 2 (one thread initializes the value at 1 and the other increments it to 2). Thus, using *ConcurrentHashMap*'s *compute* method, for instance, would fix the previous behavior (Listing 4.2).

Listing 4.1: Incorrect *ConcurrentHashMap* execution flow.

```java
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

// populate map

if (!map.containsKey("test")) {
  map.put("test", 1);
} else {
  map.put("test", map.get("test") + 1);
}
```

Ultimately, several Thyme-Infra critical sections have been analyzed during the development phase in search of potential issues, like the previous example, and adopted a more sound atomic approach.

Listing 4.2: Sound *ConcurrentHashMap* execution flow.

```java
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

// populate map

map.compute("test", (key, value) -> value == null ? 1 : value + 1);
```

Furthermore, in Thyme-Infra's *MobileNodesNetwork* and *InterInfrastructureNetwork* classes, which act as the communication layers and points of entry to clients' and infrastructure nodes' requests, each of them has an inner *ThreadPoolExecutor* [49] in order to provide a working thread to execute the messages' associated computation. By having distinct *ThreadPoolExecutors*, the system architect is able to parameterize each one individually (Section 4.9.9), with respect to the number of threads, according to the system and hardware needs: more threads for the *MobileNodesNetwork* if *AP* is expected to have a lot of clients and fewer threads for the *InterInfrastructureNetwork* if the system is composed of only a few infrastructure nodes.

4.9.2.1 Living in a World Without Infrastructure

As explained in Section 4.4.2, an instance of our updated version of Thyme starts its pre-join world process by sending a world discovery packet in multicast to the nearest infrastructure node that is able to process it, i.e. an instance of Thyme-Infra running in the infrastructure. Simultaneously, the Thyme system will also broadcast "hello" messages to its neighbors with the same intent of inquiring the currently ongoing worlds (Section 3.6), whether using the same technology (Wi-Fi) or other peer-to-peer technologies (like Bluetooth or Wi-Fi Direct). During this procedure, if Thyme does not receive any reply from a capable infrastructure node after a specified number of broadcasted retries, or if the device's user decides to integrate one of the local worlds received by its peers, the Thyme instance will fallback to a peer-to-peer only mode, essentially converting to the same behavior as Thyme's pure classic interactions, without activating any of the infrastructure-related logical components to save battery and computing power.

We are able to do this due to the extensive initial planning for both systems where we tried to idealize a solution that was as less disruptive as possible to the already developed classes and interactions by employing new features and logic in a purely additive manner. Furthermore, as previously seen in Section 4.3, the former version of Thyme used a *GeographicalRouting* class to process the respective P2P routing mechanisms while in our proposed iteration the routing is divided in two classes, the *GeographicalRouting* and the *LogicalRouting*, and each one of these is a descendant of a single *Routing* abstract superclass that is used in all of the components of the Thyme system. Thus, depending on the interaction mode and resource availability (peer-to-peer or infrastructure-aided), the system will instantiate the appropriate geographical or logical routing class for all its components.

4.9.3 Retrieving Local Popular Items to Cache in the Infrastructure

Initially described in Section 4.5, each Thyme-Infra instance periodically checks its *localPopularityInformation* table, which contains references to each object downloaded by its local clients as well as the number of retrievals, decides the most popular items for each world, sends download requests to those items and then periodically batch-inserts the received items in the *Local Popularity Cache*. This process of Retrieving Local Popular Items to Cache (RLPIC) in the infrastructure is composed of two different timers to coordinate the periodic mechanisms: popular decision and batch-insert timers (Fig. 4.17).

Popular Decision Timer (PDT). After the arrival of the first *cell data* message, the system will schedule this timer for repeated fixed-rated executions, with regular intervals between them, after a specified delay (this value is configurable by the administrator, see Section 4.9.9). When this timer fires, the process will always increment the global RLPIC counter but will purposely skip the first execution in order to give the system and its clients enough time to create meaningful and considerable data before checking the popularity of the published items to offload to the infrastructure. On further executions, the Thyme-Infra node will go through its *localPopularityInformation* table, determine the top most popular data among its local clients (Section 4.9.3.1) and broadcast a download message to each individual items' replicas. This RLPIC counter represents the current popular decision cycle and is mainly used to guarantee that the items requested within one popular decision cycle arrive before the next one. To check for this, when the infrastructure node sends a local download request as part of this process, it also associates to that request the current counter. For instance, if the current RFLPIC counter is 10 and one of the requested items arrive at the infrastructure with the counter also at 10, that item will be accepted and saved to be batch-inserted in the near future. On the other hand, if the current RLPIC value is 12 and an incoming item has 10 in its associated cycle counter, meaning that the replica took too long to process the infrastructure request due to being busy or resource-throttled (for instance), it will be ignored by the Thyme-Infra instance since its popularity index could have changed between cycles 10-12 and no longer considered popular and worthy of cache space.

Batch-Insert Timer. After the execution of each Popular Decision Timer's cycle, another timer, the Batch-Insert timer, will be scheduled to fire in a predetermined amount of regular-intervaled executions (values also configured by the administrator - Section 4.9.9) until the next PDT activation. This timer has the goal of batch-inserting already received items that were requested by the previous process into the *Local Popularity Cache* and trigger the execution of the cache's *Adaptive Multipart Caching* algorithm, as explained in Section 4.7.2. We believe that this batch-insert approach offers the best benefits both in terms of user experience and performance. The other two considered alternatives were:

1. Inserting individual items into the cache as they arrive at the infrastructure, which would make entries more readily available for the *AP's* clients but would have a considerable impact on performance since the *AMC* would have to be executed for every single insertion and, depending on its properties, individual puts could have minimal to no changes on the outcome of Equation 4.1;

2. Inserting all received items right before the next execution of the Popular Decision cycle which would have a lower impact on performance since the *AMC* would be executed only once per cycle but the content would take longer to be available to the users, degrading the user experience linearly with respect to the PDT's interval delay.

Finally, the scheduling of all the Batch-Insert Timer's intervals, between popularity decisions, takes into account what we call a "Processing Slack", γ, at the very end with the goal of giving enough time for the Thyme-Infra instance to finalize the insertion process to the *Local Popularity Cache* before starting the next cycle. Thus, avoiding scenarios where a specific item is about to be inserted into the cache and an overlapping Popularity Decision process determines that the exact same item is not yet cached and still considered popular and proceeds to ask the replica again for the data, resulting in an unnecessary duplicate request.

4.9.3.1 Determining the Top Most Popular Data to Offload to the Infrastructure

In order to choose the most relevant items to offload to the infrastructure during every popularity decision cycle to be cached in the Local Popularity Cache for easier access, each Thyme-Infra instance executes a custom filtering algorithm to establish a ranking order of popularity. Among all of the state-of-the-art projects developed in this area that were studied during the context of this dissertation (Chapter 2), we decided to focus on approaches that were computationally simple and fast while still offering relevant

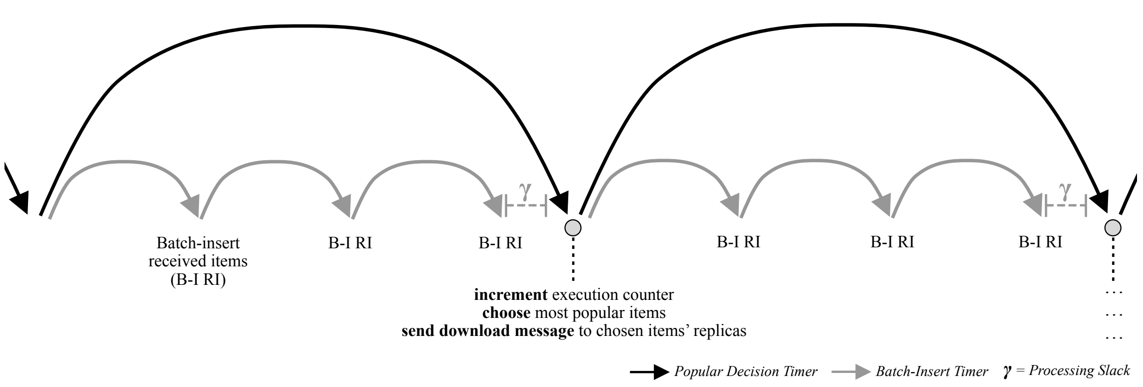

Batch-insert received items (B-I RI)
B-I RI
B-I RI
B-I RI
B-I RI
B-I RI
γ
γ
increment execution counter
choose most popular items
send download message to chosen items' replicas
...
Popular Decision Timer
Batch-Insert Timer
γ = Processing Slack

Figure 4.17: Timer definition for the process of retrieving local popular items to cache in the infrastructure.

results. The reasoning behind this is due to the fact that this process will need to be executed periodically and we do not wish to overwhelm the infrastructure node with complex computations. Furthermore, due to the periodicity of this decision process, we are able to use simpler algorithms, which would tend to have lower accuracy as a tradeoff, since we are expecting to eventually converge to the best items. Thus, considering its characteristics and positive results, we implemented a simplified approach of [31] by eradicating the authors' use of computationally intensive file clustering algorithms and incorporating Thyme-Infra-specific features and steps. Our vision for the popularity decision algorithm is composed of the following stages:

1. We start by splitting the items from the *localPopularityInformation* table into two distinct groups: active files (files that have been accessed/downloaded recently) and stale files (files not accessed in the most recent time period). In [31], the authors use the exact time, right down to the milliseconds, of when an item was downloaded in order to check whether it is within the active files' time window. In Thyme-Infra, we use a coarser granularity approach to time, in order to achieve a better (lower) space complexity during the execution of this process: whenever a Thyme-Infra node receives a *cell data* message, it will set the current RLPIC counter value to the items that were marked as downloaded in the message as their latest download timestamp. Therefore, the infrastructure node will consider active files as the ones that have been accessed in the past $active_{\text{window}}$ (value configured by the developed - see Section 4.9.9) popular decision cycles, thus passing the following condition:

$$0 \leq RLPIC - item_{\text{downloadTs}} \leq active_{\text{window}}$$

where $item_{\text{downloadTs}}$ is the last RLPIC counter associated with the item;

2. Secondly, we take the set of all active files, divide them by world and within each world's container we will sort the items further by popularity;

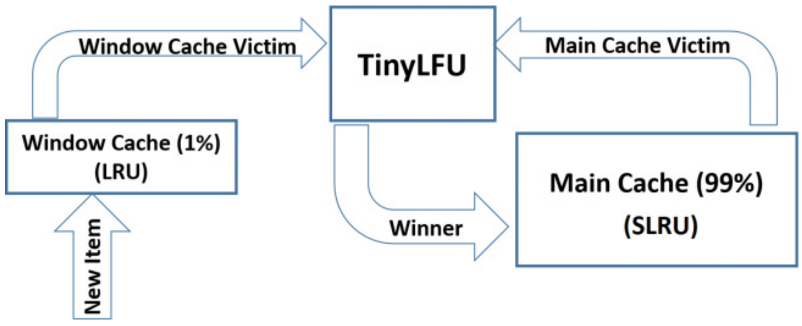


Figure 4.18: Caffeine's Window TinyLFU scheme. Adapted from [52].

3. Then, for each world container we calculate the sum of the popularity of all the inner items while also calculating the total amount of downloads throughout every world. With these values obtained, we will then compute the weight of each world based on its contribution to the total amount of downloads executed within the *BSS*;

4. Finally, we will use the upper bound of the number of items that should be considered popular during each cycle (Section 4.9.9), and then choose the top N items from each world, with regards to their calculated weights, until that limit is reached.

By the end of the execution of this algorithm, Thyme-Infra will possess the identifiers of the most popular items from this cycle and is now ready to communicate with its local replicas to download and offload them to the infrastructure. Further iterations of this algorithm will skip items that are considered popular but are already cached within the *Local Popularity Cache*.

4.9.4 Caffeinated Data

All of the Publish/Subscribe caches used by Thyme-Infra are built on top of Caffeine. Caffeine [50], developed by Ben Manes, is a high performance, thread-safe and near optimal caching library based on Java 8, which offers features such as size and time-based eviction policies, notification of evicted entries, ability to propagate writes to external sources (such as a cloud-backed database) and the automatic accumulation of cache access statistics.

Unlike simpler and widely used eviction policies like the LRU [51], which keeps track of the ordering of accesses to then choose the entry that was last obtained to be removed, and the LFU, which on the other hand keeps track of the number of accesses (or frequency) of each entry and chooses the entry with the least amount to be removed, Caffeine employs a novel and a more complex approach with the proposal of the Window TinyLFU (W-TinyLFU). W-TinyLFU [52] is a frequency-based cache admission policy which consists of three essential cache areas: an admission window, the TinyLFU admission filter and the cache itself (Fig. 4.18).

During the put operation, the new item is inserted into the W-TinyLFU's admission window which will give it a chance to build up its popularity to be able to persistently move to the main cache in the future. When this window is full, a window victim is chosen, using the LRU policy, and passed on to the TinyLFU admission filter. This TinyLFU filter, which maintains an approximate representation of the access frequency — the popularity-sketch — of a large sample of recently accessed items, will then decide, based on the recent access history, whether it is worth admitting the window's item item into the main cache, i.e. if the new item is expected to increase the hit ratio, at the expense of the eviction of an already stored item. To do this, the TinyLFU estimates and compares the newest and evicted items' frequency and recency values, using methodologies similar to the Counting Bloom Filter [53] and the Count-Min Sketch [54], and if the former's is indeed higher, the winner will me transferred to the main cache, excluding the recently evicted item to make room for it.

On the other end of the spectrum, the main cache is based on the Segmented LRU policy (SLRU) [55] which, in itself, means that the cache is divided into two parts: the protected and the probation segments, usually using 80% and 20% of the cache's size, respectively. An entry starts in the probationary segment and on a subsequent access it is promoted to the protected segment and when the protected segment is full it evicts into the probationary segment, which may trigger a probationary entry to be discarded. Furthermore, when the TinyLFU executes the described comparison process, the chosen main cache's item is retrieved from the probation container.

When compared to other high level Java in-memory caches, like Google's Guava [56] and Ehcache [57], extensive real-world and simulated benchmarks show that, regardless of the workload, Caffeine offers a better read, write and read/write performance (ops/s) [58] [59] as well as a lower memory overhead [60] while, on the other hand, when compared to other popular eviction policies Caffeine's W-TinyLFU presents a comparable or a higher, near optimal, hit rate value [61].

Adding to all of this, the fact that W-TinyLFU does not necessarily remove an item from the cache during an insertion of another, something that is employed by LRU and LFU, is highly beneficial. For instance, if we consider a cache where all the items are highly popular and the difference between the number of accesses of every one of them is negligible, an approach like LRU or LFU would have no problem in evicting a popular item to make space for another, even if the latter is not expected to be nowhere near as popular as the former, something that would not easily happen with W-TinyLFU. Thus, when we acknowledge its features and flexibility, we decided that using Caffeine in Thyme-Infra would best fit our needs.

4.9.5 Mobile Address Translation Store

As defined by the authors of Thyme [18], the system utilizes a previously built peer-to-peer communication layer, developed by other members of the *"Hyrax: Crowd-Sourcing* *mobile devices to develop edge cloud"* project [35]. By using this layer, each Thyme node has its own logical address, composed of a *UUID* [36] and the cell identifier where the client lies in, like the following:

$$(UUID : idCell)$$

such as

$$(123e4567-e89b-12d3-a456-426655440000 : 0)$$

These addresses are used by the communication layer to allow the intercommunication of mobile nodes within the system. The *UUID*'s address counterpart is randomly assigned when a Thyme instance boots up and is not changed throughout its execution. On the other hand, the cell identifier is set to the value given by the infrastructure and will essentially remain the same after that, if the system is using the nearest infrastructure nodes, otherwise the $idCell$ will be changed every time the client physically moves from one cell to another.

Logical addresses are ubiquitous to the entire workflow of the Thyme system, adopted not only to allow P2P communication between clients but also used in places such as metadata items, namely in the *replication list* field (L_{rep} - Section 3.3), which holds clients' logical addresses that are currently replicating the metadata's object (used for the download operation, for instance).

Since this layer relies heavily on mobile libraries, namely Android [19] ones, it was not possible to use it directly in the infrastructure layer, which is built in plain Java [62], in order to facilitate the connection between the Thyme and Thyme-Infra nodes. Thus, we defaulted to using the simplest communication approach by using physical addresses (IP addresses and ports) during infrastructure-to-infrastructure communications, as well as infrastructure-to-mobile transmissions. Consequently, we needed to employ a mechanism that not only allowed Thyme-Infra instances to read and comprehend Thyme's logical addresses but also transform them into physical ones for communication. To do that, we created the *Mobile Address Translation Store*, or *MATS* for short.

MATS is a single-purpose one dimensional cache that maps logical address into physical, usable, addresses for the infrastructure layer. *MATS*, built on top of the already described Caffeine cache, is size and time-bounded (both aspects can be entirely configured - Section 4.9.9): each entry has a time-to-live (since mobiles nodes can become inactive or leave the venue's coverage) and whenever the infrastructure executes a put operation of an address that is already in *MATS*, the entry's TTL will be refreshed since it is proof that the device is still operational. Furthermore, the *Mobile Address Translation Store* also uses an additional structure, a simple set, that is used to store the latest auto-elected stable node for each of the available cells. This particular data structure is used during various communications workflows as will be described in Section 4.9.7.

To populate this cache, every time the Thyme-Infra node receives an incoming message from a mobile node it will then insert into *MATS* the node's logical address as well as its physical address, obtained from the opened Socket. To obtain the former address, we altered all messages' Protobuf definition file (Section 4.9.1) that are sent from the mobile nodes to the infrastructure, with the goal of adding an *origin* field which will contain the node's logical address.

Finally, the Thyme-Infra node will then use the *MATS* when it needs to establish a direct communication channel with one of its clients, whether it is to perform a download operation on one of a metadata's replica nodes or to notify the arrival of data from other *APs*. This process will also be described in full detail in Section 4.9.7.

4.9.6 Mobile Nodes Notification on Nearest Infrastructure's Storage Changes

As defined in Thyme's updated subscription workflow specification, the system constantly checks whether a metadata item is stored in the nearest infrastructure node to optimize and structure its operations behaviors. However, at any given time, a mobile node does not necessarily know which items are stored in the Thyme-Infra node's caches beforehand. To allow this, we first extended Thyme's metadata object, even further than what we have described in Section 4.4.3, to accommodate an additional boolean field, namely *isInNearestInfrastructure*, that indicates whether the metadata object itself is stored in the nearest *AP*. Secondly, whenever an entry is inserted or removed from one of Thyme-Infra's caches, the system will send an *InfraStorageUpdate* message to the cells that are currently indexing the added/evicted item in order for the mobile nodes located within them to be able to update their own metadata's flags. This message is simply composed of a boolean flag, where *true* signals that the metadata item is now stored in the infrastructure and *false* when that item was evicted from one of the caches.

4.9.7 Infrastructure-to-Mobile Nodes Communication Workflows

To communicate with its mobile nodes, a Thyme-Infra node needs to employ essentially two types of communication workflows, depending on the operation that needs to be executed. Such workflows are: communication with a specific node and interaction with a logical cell as a whole.

4.9.7.1 Communicating With a Specific Node

To communicate with a specific mobile client, an infrastructure node employs a twofold approach as visible in Fig. 4.19. First, it inquires *MATS* to try and translate the node's logic address (represented as *nodeAddress*) into a physical one. If *MATS* indeed contains the right logical-to-physical translation, the infrastructure node directly sends the message to that destination. On the other hand, if the *Mobile Address Translation Store* is unable to

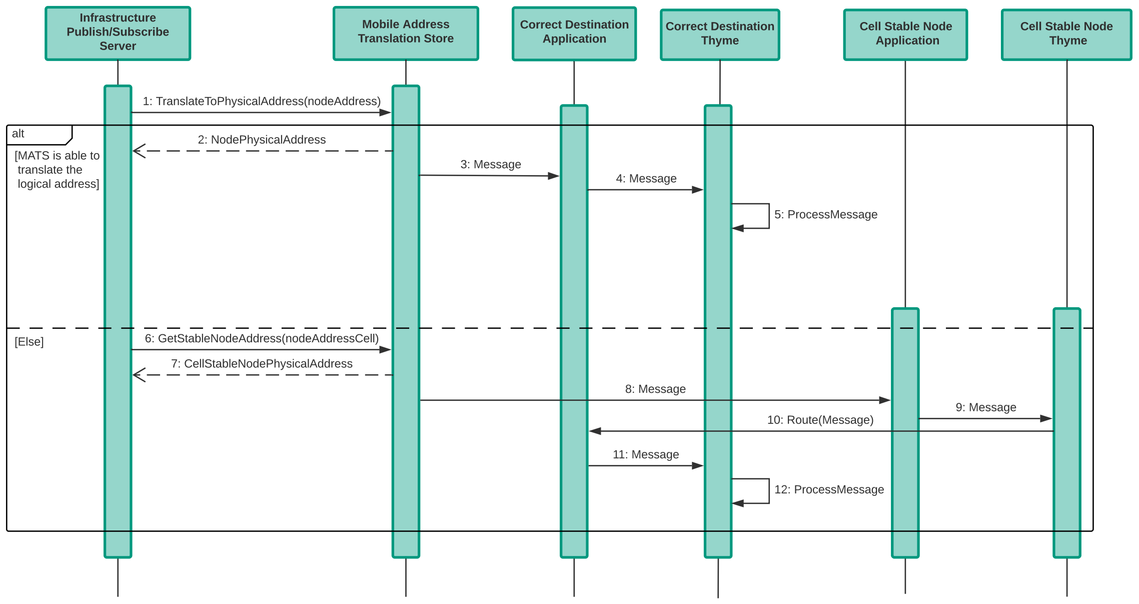

Infrastructure Publish/Subscribe Server
Mobile Address Translation Store
Correct Destination Application
Correct Destination Thyme
Cell Stable Node Application
Cell Stable Node Thyme
1: TranslateToPhysicalAddress(nodeAddress)
alt
[MATS is able to translate the logical address]
2: NodePhysicalAddress
3: Message
4: Message
5: ProcessMessage
[Else]
6: GetStableNodeAddress(nodeAddressCell)
7: CellStableNodePhysicalAddress
8: Message
9: Message
10: Route(Message)
11: Message
12: ProcessMessage

Figure 4.19: Sequence diagram showcasing the generic workflow used by the infrastructure node to communicate with a specific mobile node within its *BSS*.

process that request, i.e. due to other mobile addresses overwriting that specific node's address entry or that entry's TTL expired, the infrastructure then decides to obtain the stable node's physical address that is currently part of the same cell as the initial node (represented as *nodeAddressCell*) and sends the message to that destination which, in turn, that node will locally route the message to the correct neighbor/destination.

This particular workflow is currently only used as part of the download operation initiated by an infrastructure node, i.e. to obtain local popular items and replicate them within its local storage. Therefore, the entire workflow presented in Fig. 4.19 will be executed for each replica until the item is successful obtain from one of them. Consequently, the messages sent during this operation are *DownloadMessages* and the *ProcessMessage* steps returns the requested object or a failure message back to the infrastructure.

4.9.7.2 Communicating With a Cell as a Whole

This particular workflow, executed during the arrival of *SubscriptionNeedsReplyMessages* from other infrastructure nodes as well as part of the infrastructure's storage changes notification mechanism, was implemented to facilitate the communication and dissemination of information to all nodes within a cell by the infrastructure. The simplified version of this procedure is visible in Fig. 4.20. In short, the infrastructure node starts by asking *MATS* for the physical address of the specified cell stable node, then sends the cell message to that mobile node which, in turn, will locally disseminate the same message to all other nodes within its cell (process referred as *ProcessCellMessage* in the figure). However, if *MATS* does not know the physical address translation for the stable cell node or if the communication between the infrastructure and the stable node failed, the workflow

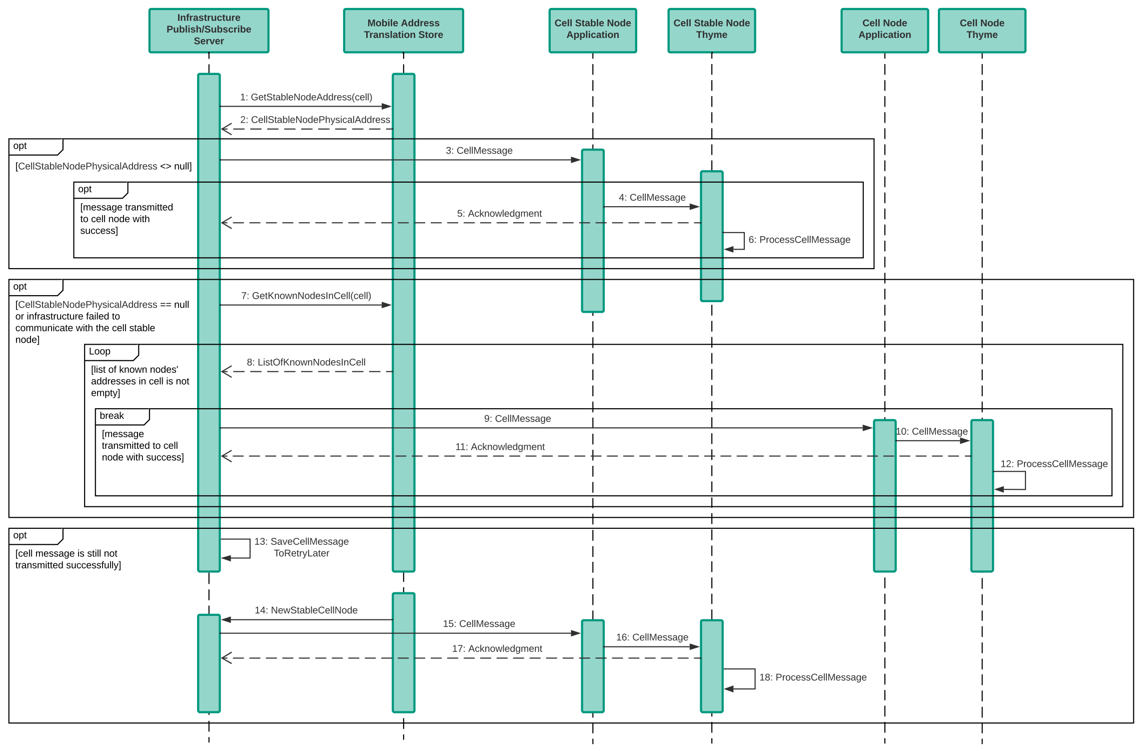

Infrastructure Publish/Subscribe Server
Mobile Address Translation Store
Cell Stable Node Application
Cell Stable Node Thyme
Cell Node Application
Cell Node Thyme
1: GetStableNodeAddress(cell)
2: CellStableNodePhysicalAddress
opt
[CellStableNodePhysicalAddress <> null]
3: CellMessage
opt
[message transmitted to cell node with success]
4: CellMessage
5: Acknowledgment
6: ProcessCellMessage
opt
[CellStableNodePhysicalAddress == null or infrastructure failed to communicate with the cell stable node]
7: GetKnownNodesInCell(cell)
Loop
[list of known nodes' addresses in cell is not empty]
8: ListOfKnownNodesInCell
break
[message transmitted to cell node with success]
9: CellMessage
10: CellMessage
11: Acknowledgment
12: ProcessCellMessage
opt
[cell message is still not transmitted successfully]
13: SaveCellMessage ToRetryLater
14: NewStableCellNode
15: CellMessage
16: CellMessage
17: Acknowledgment
18: ProcessCellMessage

Figure 4.20: Sequence diagram showcasing the simplified workflow used by the infrastructure node to disseminate messages to all nodes within a cell in its *BSS*.

will try further by requesting the translation store for all known physical addresses of nodes in that cell and iteratively execute the same procedure as described above until it the transmission succeeds. Ultimately, if all of these methods fail, the infrastructure will put the message dissemination on hold, wait for the *Mobile Address Translation Store* to notify it when another stable node is established for that particular cell and, when that notification is triggered, the dissemination will be retried once again.

4.9.8 Time outs

In order to keep the processing of critical operations as snappy as possible, such as downloads and subscriptions that, while performed asynchronously and not considered UI-bound, the user still expects a visual feedback after executing these operations, we employ timeouts in all messages sent across the system, whether by mobile or infrastructure nodes, with the goal of avoiding scenarios where mobile or even infrastructure nodes may be down or overwhelmed by requests at a given time. Since the number of retries and the timeouts between each one of them rely heavily on the hardware available within the venue, as well on the venue's network specifications, we believe that those values should not be an hardcoded one-size-fits-all solution. Thus, we give the option to the architecture developer to fine-tune this controls and choose the most appropriate values in the Thyme-Infra configurations file (Section 4.9.9).

4.9.9 Configurations File

One of the goals throughout the entire development of Thyme-Infra was to create a system that would be able to adapt to every scenario and would be able to be fine-tuned by the infrastructure/application developer to tailor to not only its clients/applications needs but also to its hardware needs. Therefore, we created a configuration file that is required by the system during startup whose goal is to provide the infrastructure application multiple details on how it should run, whether it is timeout values, cache sizes, etc.

Table 4.1 presents every entry available in the configurations file as well as a brief description to each one, along with a distinction between required and optional values. The following represents a concrete example of a possible configurations file:

Listing 4.3: Thyme-Infra example configurations file.

```
infra.own.ip = 192.168.1.67
infra.own.mobile.port = 9500
infra.own.mobile.nThreads = 10
infra.own.interinfra.port = 9501
infra.own.interinfra.nThreads = 5

infra.own.discoveryMulticast.ip = 227.7.12.95
infra.own.discoveryMulticast.port = 8999

infra.own.cache.localPopularity.size = 100
infra.own.cache.prefetch.metadata.size = 101
infra.own.cache.prefetch.data.size = 102
infra.own.cache.global.metadata.size = 103
infra.own.cache.global.data.size = 104
infra.own.cache.mobileAddresses.size = 100
infra.own.cache.mobileAddresses.writeTTL = 300000

infra.own.subNeedsDisseminationDelay = 60000
infra.own.subNeedsReplyDisseminationTimeWindow = 60000
infra.own.retrieveLocalPopItemsToCacheDelay = 30000
infra.own.maxLocalPopularItemsToOffload = 100
infra.own.popularItemsActiveWindow = 3
infra.own.numberOfBatchInsertsToLocalPopularityCache = 3
infra.own.localNodesRequestsTimeout = 3000
infra.own.infraRequestsTimeout = 3000

infra.own.popularityEntryMaxStorageLocations = 10
infra.own.localDataGloballyConsumedInitialIgnoreCounter = 5

infra.own.usesPopRankingAlgorithm = false
infra.own.usesConsumptionNotification = false

mobiles.inSimulationEnvironment = true

infra.other.0.ip = 2.2.2.3
infra.other.0.port = 9502

infra.other.1.ip = 2.2.2.4
infra.other.1.port = 9603
```

To enable the system to read and parse the configurations file, the system administrator should execute the Thyme-Infra instance while specifying the following Java Virtual Machine argument, *-Dconf.file*, with the appropriate configurations' file path, like so:

```
$ java -Dconf.file=<conf-file-path> -jar <thyme-infra-path>.jar
```

Table 4.1: Thyme-Infrastructure Configurations File Specification. **Bold** entry keys represent optional values.

Key		Description
infra.own.		
	ip	The IP address of the current infrastructure node
	mobile.port	The current infrastructure node port used to receive messages from the mobile nodes
	mobile.nThreads	The number of threads to be allocated to process incoming requests from the mobile nodes
	interinfra.port	The current infrastructure node port used to receive messages from other infrastructure nodes
	interinfra.nThreads	The number of threads to be allocated to process incoming requests from the infrastructure nodes
	discoveryMulticast.ip	The multicast IP address for the mobile nodes to discover the current infrastructure node
	discoveryMulticast.port	The multicast port for the mobile nodes to discover the current infrastructure node
	cache.localPopularity.size	Local Popularity Cache max number of entries
	cache.prefetch.metadata.size	Prefetch cache max number of metadata entries
	cache.prefetch.data.size	Prefetch cache max number of data entries
	cache.global.metadata.size	Global cache max number of metadata entries
	cache.global.data.size	Global cache max number of data entries
	cache.mobileAddresses.size	Mobile Nodes Address Store max number of entries (as explained in Section 4.9.5)
	cache.mobileAddresses.writeTTL	Mobile Nodes Address Store entry TTL after a write/put (as explained in Section 4.9.5)
	subNeedsDisseminationDelay	How long, in ms, should the Thyme-Infra node wait between its own subscription needs disseminations (as explain in Section 4.6)
	subNeedsReplyDisseminationTimeWindow	How long, in ms, is the Thyme-Infra node's non-overlapping subscription needs reply dissemination time window (as explain in Section 4.6.1)
	retrieveLocalPopItemsToCacheDelay	Delay, in ms, between popularity decision cycles
	maxLocalPopularItemsToOffload	The max number of items to be considered popular and offloaded into the infrastructure during each popularity decision cycle (as explained in Section 4.9.3.1)
	popularItemsActiveWindow	Value that represents the window of which items need to have been accessed to be considered as "active"
	numberOfBatchInsertsToLocalPopularityCache	The number of batch inserts between popularity decision cycles
	localNodesRequestsTimeout	Timeout, in ms, before a request to a mobile node is considered invalid
	infraRequestsTimeout	Timeout, in ms, before a request to another infrastructure node is considered invalid
	popularityEntryMaxStorageLocations	Maximum size for the data structure which keeps track of each item's storage locations (or replicas) in the local popularity information table
	localDataGloballyConsumedInitialIgnoreCounter	The number of dissemination cycles to be executed before being able to start disseminating a specific item again after it was already consumed (as explain in Section 4.6, Step 3)
	usesPopRankingAlgorithm	Flag used to indicate whether the popularity ranking algorithm should be executed to decide which items should be offloaded into the infrastructure during each popular decision cycle. Defaults to *true*
	usesConsumptionNotification	Flag used to indicate whether the Thyme-Infra instance should notify an item's origin infrastructure node when it is consumed by its local users. Defaults to *true*
mobiles.		
	inSimulationEnvironment	Flag used to indicate whether the Thyme nodes communicating with the infrastructure are in a simulation environment (i.e., within a single machine) in order to adapt some components to work with the simulation. Defaults to *false*
infra.other.$\{n : n \in \mathbb{N}\}$.		
	ip	The IP address of another infrastructure node from the system
	port	The port of another infrastructure node from the system

4.10 Summary

In this chapter we presented the updated Thyme model as well as the proposal and the implementation of the novel Thyme-Infra component. Furthermore, we described each individual layer of the entire system, alongside a characterization of its behavior and associated outcome, as well as thorough explanation of the implementation details and decisions that had to adopt in order to create a solution that is as performant and sound as possible in the context of edge networks. After reading this chapter, one is able not only to comprehend how the proposed scheme works on a higher level but also how each low-level component behaves and interacts with the rest of the system.

Chapter 5

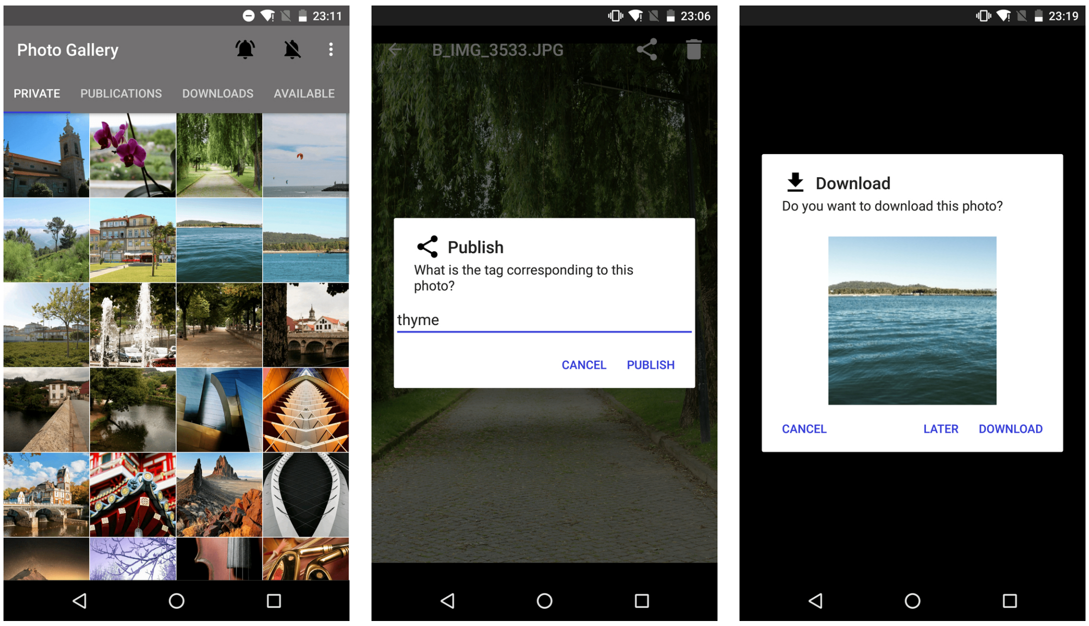
Evaluation

In this chapter we will showcase the results of our extensive experimental evaluation in order to characterize our proposed system, in terms of behavior and performance. In Section 5.1 the evaluation methodologies are presented, along with an overview of the use case application that acts as the core of our entire testing infrastructure. To guarantee the logical soundness of the proposed solution, we will present, in Section 5.2, our logical evaluation. In Sections 5.3 and 5.4 we will demonstrate how our system performs, under a variety of tracked metrics, as well as how it stacks up to the previous version of Thyme, respectively. Finally, in Section 5.5, the combination of Thyme and Thyme-Infra will be directly compared to a previously deployed cloud infrastructure.

5.1 Evaluation Methodologies

To evaluate the entire behavior and workflows of our proposed solution, we segmented the testing scenarios into two distinct environments: real-world and simulated environments.

Real-world Evaluation. Thyme and Thyme-Infra can be used to implement a plethora of use cases and applications that run at the edge. One specific use case would be a photo gallery application for Android devices that allows the user to share photos between mobile devices interconnected through a wireless network. A photo gallery software application was initially developed by the authors of Thyme [18] (Fig. 5.1) to serve as way of showing how Thyme works and how well it performs in real-world scenarios. This *app* supports all regular Publish/Subscribe operations, like the subscription, download, and publish.



Publish

What is the tag corresponding to this photo?

thyme

Download

Do you want to download this photo?

a Main Screen. b Publish Item Screen. c Download Photo Screen.

Figure 5.1: Showcase of the use case application. Adapted from [18].

In the context of this dissertation, we will bootstrap our testing and evaluation environment by utilizing the already developed application to collect metrics for the newest iteration of Thyme and record the impact of the introduction of the Thyme-Infra layer in the system. Furthermore, with this evaluation methodology, we focused on harvesting as much data as possible regarding physical-related metrics, in the likes of battery consumption and transmissions latency, which cannot be accurately measured in a simulated scenario.

To perform this real-world assessment, we utilized a variety of devices, namely Android smartphones, from an array of specific brands with their respective hardware properties as seen in Table 5.1. Nonetheless, due to hardware limitations, we were only able to gather a set of 6 physical devices to pursue our evaluation needs.

Table 5.1: Mobile devices used for the real-world evaluation scenarios.

Device	**Motorola Nexus 6**	**Motorola Moto G (2nd gen)**
CPU	Quad-core 2.7 GHz Krait 450	Quad-core 1.2 GHz Cortex-A7
RAM	3 GB	1 GB
Storage	32 GB	8 GB
Battery	Li-Po 3220 mAh	Li-Ion 2070 mAh
OS	Android 7.1.1 (Nougat)	Android 7.1.1 (Nougat)
Wi-Fi	Wi-Fi 802.11 a/b/g/n/ac	Wi-Fi 802.11 b/g/n
Bluetooth	4.1	4.0

Simulated Evaluation. Regarding the simulated environment, we utilized a custom trace-based simulation framework, previously developed in-house in pure Java, which offers a total rework of Thyme's networking layer to support logical dissemination of messages between any number of virtual nodes within a single machine. To support this type of evaluation, we had to adapt a considerable amount of Thyme's logic, such as exchanging Android-specific classes and dependencies with their simulated counterparts. Furthermore, by executing our set of testing scenarios in a simulation environment, we are able to test, in greater detail, the scalability of the proposed solution without any hardware limitations. Concerning the Thyme-Infra counterpart, since we designed it to be a pure Java component from the ground up, we did not need to alter it in any way to be used within the simulator. Therefore, we just need to execute the Thyme-Infra instance and link all the Thyme simulated nodes to it in order to establish communication channels between both parties.

Unlike the real-world evaluation where we used a photo sharing application as the basis of our testbed, in the simulation environment we specifically created custom traces that would mimic a real-world scenario. This particular scenario is based in a university setting, where half of the users attend a class in the beginning of the trace, followed by an intermission and, after that, the rest of the students attend the second daily lecture of the same class. During each lecture, 15% of the attending students are considered the "good students" and therefore subscribe to all of the topics that will be lectured right at the beginning (known as the "class warmup interval") and are more predisposed to publishing items throughout the entire duration of the class. The rest of the students will have a lower interaction rate and will only start to contribute to the system after a predetermined amount of time into the class. Throughout the entire duration of the trace, all users will be subscribing and publishing items with tags that are relevant and global to the entire university.

Each class has a 45 minutes duration while the intermission is 10 minutes long, culminating in a total duration of 100 minutes. In the lectures, the warmup interval takes 10 mins. For the trace generation, subscription operations for most of the class students were issued randomly throughout the class duration until all of the class tags were met. Publish operations were generated with a rate of 30 and 5 per user per hour for good and regular students, respectively. Regarding university related tags, subscriptions were generated with a rate of 10 per hour, for all students, while publish operations were delivered with a rate ranging from 10, for students outside the class, and 2 per hour, for students currently attending a class. To further simulate real-world interactions and reaction times, downloads are issued with a 70% probability and within 30 seconds of the new file notification arrival. Moreover, we generated and focused on trace files with 100 virtual nodes as we believe that number to be an acceptable population size for a single access point.

To make simulation execution more lively, we compressed the ∼2 hours into 10 minutes of simulated time. Proactive routing protocols need time to converge. Thus, in our simulations, the application running on the nodes only started after 60s, and operations started being issued only after that. At the end of the simulation, nodes only shutdown 60s after operations stopped being issued in order to give enough time for the ongoing processes to be executed successfully. The total simulation time was 720s.

With the goal of evaluating the system in the most relevant way possible, regarding the Thyme-Infra counterpart, we performed an extensive research on current state-of-the-art 5G-tailored base stations, and respective CPUs, in order to parameterize our framework's specifications to match real-world hardware components. One of the processors found during our investigation was Intel's Atom® Processor C3000 Series [63] specifically designed for ZTE's [64] 5G base stations. Among all of the CPU series characteristics, the processors range from 2 cores, on the lower-end models, to 16 cores on the higher-end of the spectrum. By having this information, we decided to artificially limit the cores used by each Thyme-Infra instance to be 16, in order to grasp how the system would perform if executed in a real-world 5G equipment. Moreover, to actually simulate a physical 5G base station executing our infrastructure software stack, we utilized a computer laptop, running an instance of Thyme-Infra, directly connected to a consumer-grade wireless router (such as the Huawei EchoLife HN8245Q [65]). On top of that, to implement scenarios where multiple infrastructure nodes are available in the system, each of the routers will be physically linked by cable to each other. The technical specifications of the different laptops used for this specific purpose are displayed in Table 5.2.

Table 5.2: Computer laptops for running Thyme-Infra in the real-world evaluation scenarios.

Model	**Apple MacBook Pro Mid-2014**	**Apple MacBook Pro Mid-2010**
CPU	Dual-core 2.6 GHz Intel Core i5	2.4 GHz Intel Core 2 Duo
RAM	8 GB	4 GB
Storage	120 GB	250 GB
OS	macOS 10.14 (Mojave)	macOS 10.13.5 (High Sierra)
Wi-Fi	Wi-Fi 802.11 a/b/g/n/ac	Wi-Fi 802.11 a/b/g/n

Additionally, in order to focus, evaluate and produce a precise overview of the raw overhead produced by our system running in an edge network setting, we restricted the size of the data items published by the mobile clients to be precisely 64 bytes long in nearly all of the tests and in both of the evaluation scenarios (real-world and simulated). This allowed us to add some small weight to the messages that contain file data and create a clear distinction between them and the rest of the traffic. On top of that, for every given evaluation scenario, we reported the average results of at least 3 independent executions.

Ultimately, we seek to answer the following questions:

1. What is the impact of multiple parameterizations of the *Mobile Address Translation*

Store?

2. What is the impact on the caches as well as on the system performance while using our proposed popularity ranking algorithm versus no algorithm at all?
3. How is the system's performance impacted by using the proposed Items' Consumption Notification mechanism?
4. How does the *Adaptive Multipart Caching* system behave under distinct population needs?
5. What are the tradeoffs of utilizing a multiple batch-insert approach in the Local Popularity cache when compared to approaches like inserting the item right away when it reaches the infrastructure or inserting all the received data at once at the end of each cycle?
6. How does the previous iteration of Thyme compare to the updated Thyme along with Thyme-Infra in terms of performance and resource utilization?
7. How does the proposed edge system stack up to a full-fledged external cloud infrastructure solution?

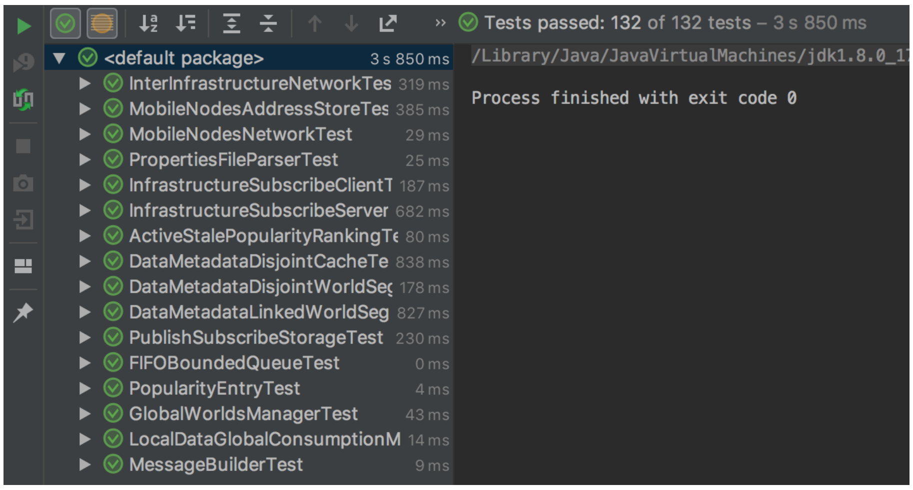
5.2 Logical Evaluation

Due to the inherent complications associated with software development, namely the occurrence of "software bugs", before we started executing our proposed system to gather performance metrics, the first step in our evaluation scheme was to make sure both of the developed architectures (the newest iteration of Thyme and the Thyme-Infra) were programmatically sound and their operations' outputs and workflows were correct. To test this, we performed a logical evaluation of our entire software stack by creating unit tests that made sure each system's components behaved like they were conceptually designed to do so.

Throughout the entire development cycle of this thesis, we created over 130 unit tests (Fig. 5.2), using the JUnit Java library [66], across all core components and their operations while trying to be as thorough as possible by designing a multitude of use cases and inputs to logically validate each module during irregular and stressful conditions. Only then, after we reached a satisfactory code coverage, we felt confident about initiating the process to track evaluation metrics since our system showed no signs of incorrect behavior.

5.3 Absolute Metrics Evaluation

The main purpose of this section is to present a comprehensive description, through plotting of multiple absolute metrics, of how our system behaves and performs during



```
Tests passed: 132 of 132 tests – 3 s 850 ms

<default package>                        3 s 850 ms
  InterInfrastructureNetworkTes          319 ms
  MobileNodesAddressStoreTes             385 ms
  MobileNodesNetworkTest                  29 ms
  PropertiesFileParserTest                25 ms
  InfrastructureSubscribeClientT         187 ms
  InfrastructureSubscribeServer          682 ms
  ActiveStalePopularityRankingTe          80 ms
  DataMetadataDisjointCacheTe            838 ms
  DataMetadataDisjointWorldSeg           178 ms
  DataMetadataLinkedWorldSeg             827 ms
  PublishSubscribeStorageTest            230 ms
  FIFOBoundedQueueTest                     0 ms
  PopularityEntryTest                      4 ms
  GlobalWorldsManagerTest                 43 ms
  LocalDataGlobalConsumptionM             14 ms
  MessageBuilderTest                       9 ms

/Library/Java/JavaVirtualMachines/jdk1.8.0_1

Process finished with exit code 0
```

Figure 5.2: Execution of all unit tests in THYME-INFRA.

distinct scenarios, workflows and workloads, focusing solely on the simulated evaluation. Further, we will also put some of the features, that were specifically proposed and implemented in THYME and THYME-INFRA with the goal of increasing the overall system performance and user experience, to the test in order prove our initial assumptions.

5.3.1 Mobile Address Translation Store

The *MATS* is one of the most important components of THYME-INFRA which essentially allows the infrastructure node to be able to communicate with its clients at will, without being limited to request-and-reply interactions initiated by the mobile nodes. This data structure is particularly used during each THYME-INFRA's popularity decision cycle due to the fact that, during this step, the infrastructure node will try to contact with the chosen items' replicas, thus requiring a translation from the replica logical address into a physical one. As previously explained, when the store is unable to translate an address, it will force the system to disseminate the message (i.e. *InfraRoutingMessage*) to a node within the destination's cell, such as the cell stable node, and ask it to locally route that packet to the desired destination.

As the *Mobile Address Translation Store* is built on top of a bounded in-memory cache, namely a Caffeine cache, whose size is fully configurable by the developer, in this evaluation scenario we aim to verify the impact of the cache's size in the traffic generated by the system when compared to the produced memory overhead. As the testing environment, we utilized a single THYME-INFRA instance.

Fig. 5.3 showcases the generated traffic results. As we can see, if the infrastructure used the *MATS* with a size of 0, meaning that it would only save each cell's stable node address, an additional ∼2300 messages would have to be transmitted in order to reach the desired destinations, at multiple stages of the trace. From that point on, the number of routing messages sent starts to decrease at an accentuated rate until the *MATS* is able to store half of the total mobile population's addresses. Afterwards, and until the full store

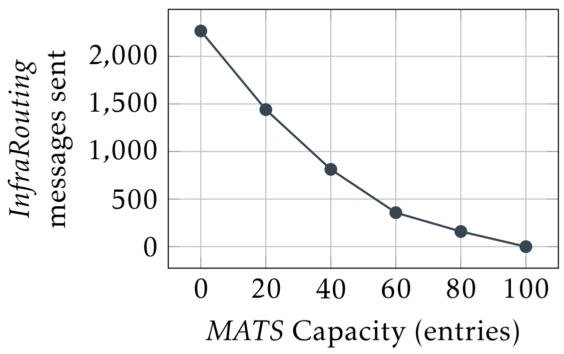


Figure 5.3: Total *InfraRouting* messages sent by the infrastructure as the *MATS* capacity increases.

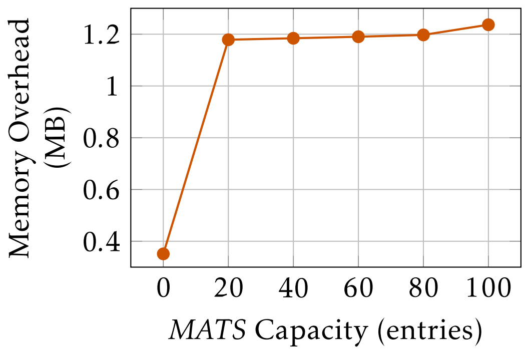


Figure 5.4: Memory overhead caused by the *MATS* as its capacity increases.

capacity is reached (which culminates in the transmission of 0 *InfraRouting* messages), the performance increases are much less noticeable.

On the other hand, in order to quantify the *MATS*'s memory usage at different sizes, we utilized the specialized VisualVM Java Profiler tool [67] which allows the user to glance at, among other things, a program's heap space. Consequently, we created a separate Java program with the intent of simply creating and populating the translation store to gather memory statistics, by applying the following steps:

1. Running the program and stopping right before the code that instantiates the *MATS*;

2. On top of that, we manually invoked the execution of the Java garbage collector. This, in turn, allowed us to snapshot and obtain the baseline of the program's heap utilization;

3. Then, we resumed the program and allowed it to create and populate the translation store;

4. Finally, we paused the process right after the previous step completed, re-executed the garbage collector once again (in order to get rid of any auxiliary and intermediate information from the memory) and obtained the heap usage, to which we deducted the baseline from the first step, of *MATS* from VisualVM.

The final results, for different store sizes, can be visualized in Fig. 5.4. With a size of zero, the *MATS* memory overhead is approximately 0.4 MB. Although in this case the inner addresses cache is not initialized, other auxiliary data structures are always instantiated when the translation store is constructed, such as the collection that keeps track of each cell's stable node address, which makes up to that amount of heap usage. When we started to increase the *Mobile Address Translation Store* we verified that by simply initializing the Caffeine cache the memory overhead would grow up to 1.15 MB, due to its intricate inner structures and tasks. Furthermore, for every additional 20 entries the usage rose by around 0.1 MB, reaching a total of 1.24 MB with the capacity for 100 mobile addresses.

To conclude, it is possible to verify that the *MATS*'s capacity has a direct impact on the performance of the system, namely the additional traffic generated, based on the total mobile population. From our perspective, in most of the times, the 1.24 MB memory overhead from a full-capacity translation store is insignificant when compared to the amount of traffic directed at mobile clients that it avoids, thus saving their battery consumption. However, if the expected population size that will join the system is considerably larger or if the deployed infrastructure node has a much more limited amount of available memory and storage resources, such as a Raspberry Pi [68] or an Arduino [69], our system is flexible enough to allow the developer to be able to control it at will.

5.3.2 Popularity Ranking Algorithm Evaluation

In Section 4.9.3.1 we introduced our take on an iteration of an edge-related popularity ranking algorithm with the goal of choosing the most relevant item to offload to the infrastructure during every popularity decision cycle to be cached in the *Local Popularity Cache* for easier access. By considering its characteristics, it is possible to presuppose some of the potential positive impacts on the performance of the system: a higher cache hit ratio on the *LPC* since only the items considered relevant by the algorithm are marked to be cached by the infrastructure; a lower transmission of download requests to the mobile nodes due to the algorithm limiting the number of items to download during each execution as well as ignoring items that, while still possessing popular traits, are no longer considered active, ultimate saving these nodes' resources.

Nonetheless, due to our role as researchers throughout the duration of this dissertation, it is imperative to put our algorithm to the test in order to collect palpable metrics to confirm the validity of our assumptions. Therefore, in this evaluation scenario, we focused on testing the popularity ranking algorithm and comparing its results to the same system execution but, this time, the Thyme-Infra node will not utilize any popularity ranking algorithm and will choose to download all items, without filter, from its local clients. Consequently, in both cases, we tracked the number of download requests initiated by the infrastructure with the goal of offloading items into its *Local Popularity Cache* as well as its hit ratio.

As a consequence of the asynchronous eviction nature of the Caffeine cache, which is used in the *LPC*, we executed the 100-nodes trace in its original duration, 720 minutes, to offset any misleading results caused by the library's approach which would be more common in the reduced simulation time:

"By default the (eviction) policy work is performed asynchronously (...). After a read or write, an operation is appended to a bounded queue and draining this queue is scheduled on an executor. For reads this occurs after a threshold while writes are immediate.

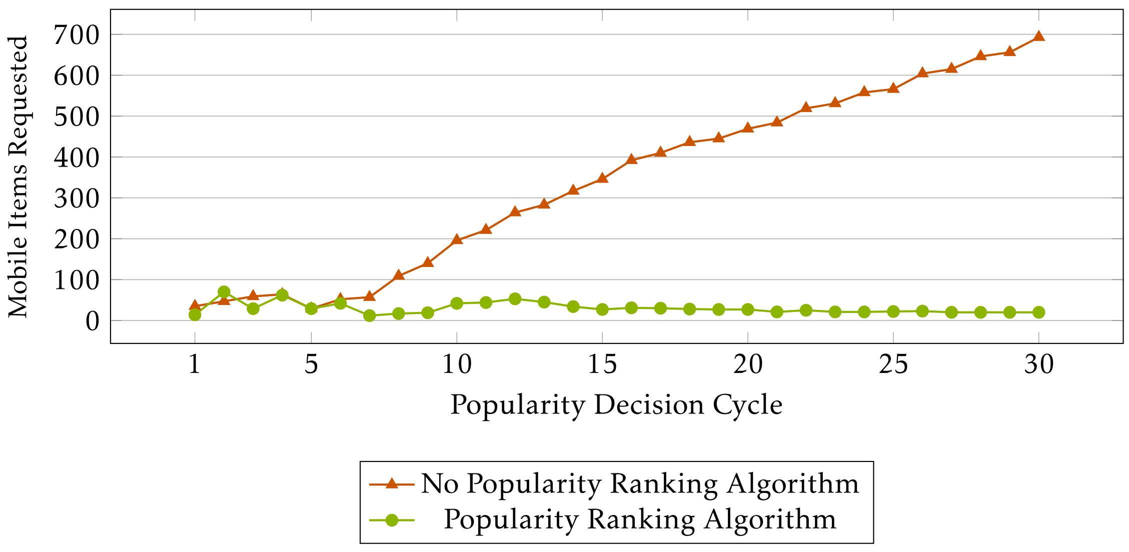


Figure 5.5: Number of mobile items requested by the infrastructure per popularity decision cycle when using a popularity ranking algorithm versus no algorithm at all.

This causes the maximum (size) to be exceeded temporarily (...).” [70]

Furthermore, we deployed a single Thyme-Infra instance and chose the first 30 decision cycles to analyze.

Fig. 5.5 presents the number of requests initiated by the infrastructure during each Popularity Decision cycle, in both scenarios, throughout the execution of the trace. While in the first few cycles the outcome is essentially the same (i.e. both are filling the cache while it is not full), subsequent cycles for the scenario without the popularity algorithm showcase, as expected, a constantly increasing amount of item requests that follows the rate of which new items are published in the network. Yet, by utilizing our proposed popularity ranking mechanism, the number of infrastructure requests starts to stabilize after the cache is full and gradually performing small adjustments to the cache’s contents throughout the trace, never requesting more than 70 items at any given cycle.

On the other hand, Fig. 5.6 shows that the *LCP* hit ratio during the scenario with the popularity ranking algorithm presents far better results, namely a 10% increase, than when not using it.

Thus, we conclude that even with our coarser approach to calculating file popularity, we are able to provide a better storage utilization — measured by the cache hit-ratio — through an acceptable amount of file requests per popularity decision cycle, when comparing to the alternative unbounded approach. We can further correlate this findings with a lower battery and processing utilization from the mobile nodes, as we are able to reduce the amount of times items are mindlessly offloaded into the infrastructure while still successfully serving 42% of the received download requests, which would have to be processed by mobile replicas otherwise.

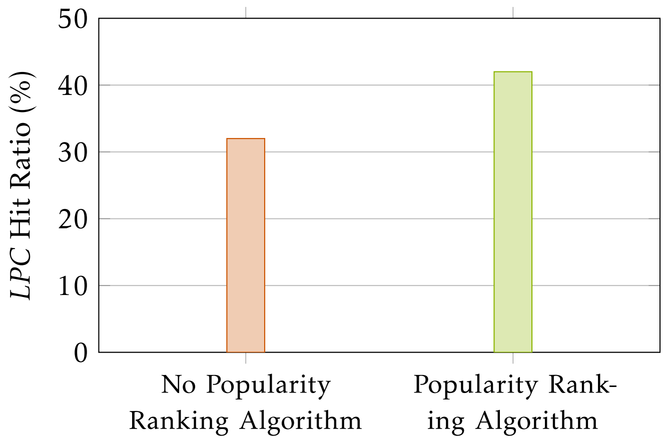


Figure 5.6: *Local Popularity Cache* hit ratio comparison between using a popularity ranking algorithm to decide which items to offload and no algorithm at all.

5.3.3 Adaptive Multipart Caching

The *Adaptive Multipart Caching* approach is at the core of Thyme-Infra in providing a more flexible caching partitioning system which accommodates to the users' needs on-the-fly, with the goal of optimizing the storage utilization of the infrastructure node.

In this testing environment, we focused on tracking the behavior of the *AMC* algorithm across multiple population scenarios to verify that it worked as it was originally intended to do so, i.e. the attributed world cache partition size matches its users interaction rate. Consequently, to create these distinct scenarios, we generated various iterations of the 100-nodes trace file and created 3 different worlds, wherein all nodes were evenly distributed across them. Furthermore, in each iteration, we altered the base rate at which publications were executed by the clients to artificially increase the popularity/intensity in a world-by-world basis to promote disparities between them.

Fig. 5.7 shows the *Local Popularity Cache* occupation (%) of each world at the end of the trace, as a consequence of the continuous execution of the *AMC* algorithm, for different popularity rates, where 1/1/1 indicates the baseline popularity for all worlds while 2/1/1 expresses that "World 1" is twice as popular than the rest of the worlds (i.e. double the number of publish operations, per hour in real-time).

As we can assert from the observed results, the *AMC* provides an equivalent size to all inner caches when each world's population presents an identical degree of popularity. On the other hand, when this equality is deteriorated and disparities are introduced into the worlds' needs, the *Adaptive Multipart Caching* is able to adapt and provide an higher cache size to the more demanding populations by reducing the size of the remaining containers as a consequence of this, for instance: the world 1 size was increased from 34% in 1/1/1 to 72% in 2/1/1 while the world 2 and 3 suffered a reduction of 24% and 16%, respectively.

Therefore, from a pragmatic standpoint, the *AMC* procedure is working as it was

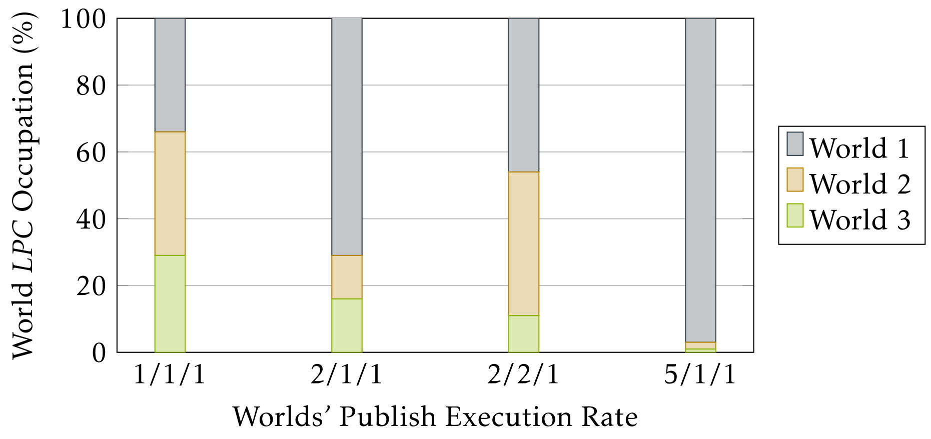


Figure 5.7: Outcome of the *Adaptive Multipart Caching* mechanism for worlds with disparate population needs.

originally intended, i.e. giving more cache real state to worlds with an increased popularity while reducing the size on the other end of the spectrum. Yet, points can be made about improving the algorithm, especially in scenarios where one of the worlds display an overwhelming popularity when compared to the rest of the system (as seen in the 5/1/1 scheme). Since the rest of the worlds are only able to amass one to two percent of the main cache's total size due to their contrastingly low needs, the partitioning system could employ additional features to tackle scenarios like this one:

- A predetermined lower bound to each world's cache size to offset the negative impacts of an overwhelming world, with the goal of ensuring that every distinct population will have a minimum guarantee that some of their items will be offloaded and served by the infrastructure;
- Another alternative implementation would approach this the other way around, by imposing a minimum amount of entries that each world needs to have before being completely evicted from the cache. This could present an acceptable tradeoff if moving the 1-2% capacity from the lowest caches into the bigger ones would result in an overall higher traffic and battery consumption reduction to the mobile nodes.

However, due to time constraints, we were not able to perform and build such supplementary tests and features.

5.3.4 Impact of the Consumption Notification Mechanism

In this section we aim to evaluate the advantages and the potential of using the proposed Item Consumption Notification mechanism in our system, namely its impact on the generated infrastructure traffic. To test this we used 2 Thyme-Infra nodes with 100 virtual clients in each one and generated two trace files that, although fundamentally different, incorporated common interests (i.e. same topics) between both clusters of

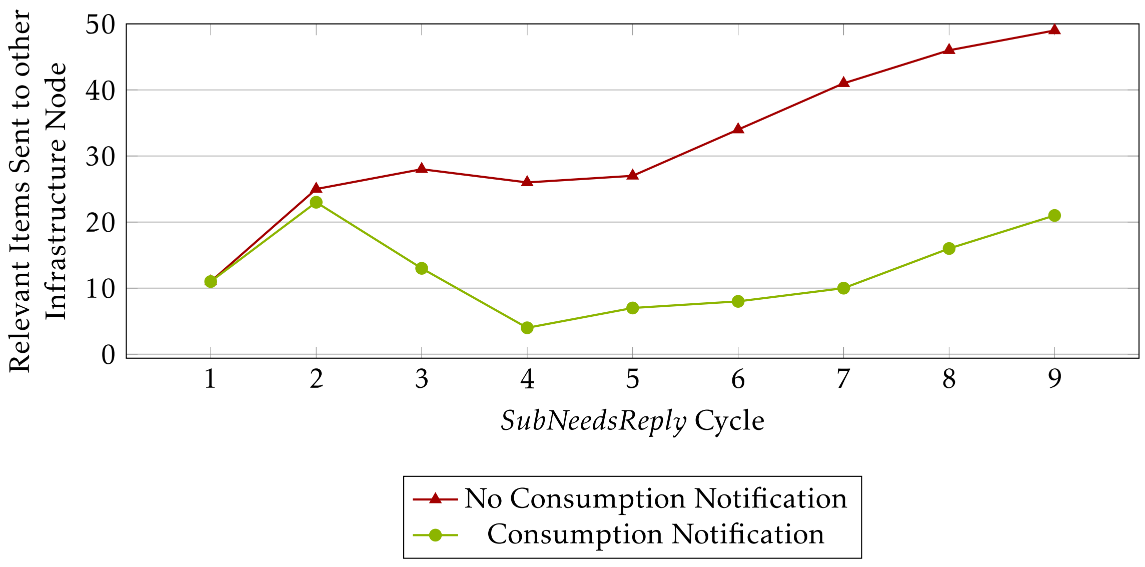


Figure 5.8: Impact on the traffic generated between infrastructure nodes when the Item Consumption Notification mechanism is enabled and disabled.

users. Fig 5.8 shows the comparison between using the notification approach versus no notification, in terms of the number of items that are sent from a single infrastructure node to another during each *SubNeedsReply* dissemination cycle.

As we can demonstrate, the number of items sent between Thyme-Infra nodes during every "Global Provisioning of Subscription Needs" step, after the first execution, is always lower using the implemented mechanism, when compared to the absence scenario. On average, this translated in a reduction of approximately 40% of objects sent per cycle. Moreover, if we consider that the published files were 1 MB in size, we would able to reduce the inter-infrastructure traffic by at least 174 MB for the entire duration of this evaluation and, as a consequence of that, avoid an unnecessary overload of the infrastructure resources.

While this mechanism provides concrete advantages, through minimizing the traffic and consequently the processing of infrastructure nodes, we believe that there are certain environments that the consumption notification would not be as useful, such as:

- Completely distinct interests between populations. Although our mechanism would provide absolutely no benefits in this situation, it would also not produce any additional overhead. This is due to the fact that the *SubNeedsReply* process itself would verify that no local items are considered relevant by the other infrastructure node's population and, therefore, would not send anything. Consequently, since no items would be transmitted between these actors, there would also not be any consumption messages being disseminated through the wired network. However, we believe the occurrence of scenarios like this one to be highly uncommon as people participating in the same event, within the same venue, are likely to have a certain degree of common interests between them;

- Populations with common interests and with an incredible amount of published data, which could cause a full turnover of the Thyme-Infra's *LocalPopularity-Caches* contents between *SubNeedsReply* cycles. Therefore, since every cycle would transmit a set of entirely new items, the consumption notification would essentially have no effect on reducing the number of duplicate items traversing between these nodes. Nonetheless, even with a setting like this one, we argue that using our mechanism could still be beneficial. Again, if we consider the published files to be 1 MB in size, since each consumption message is on average 80 bytes, it would take the transmission of 125000 of these messages ($\frac{1MB}{80\,bytes}$) to stop being advantageous. That is, if we are able to save a single file from being redundantly sent to the other infrastructure node, within that transmission limit, the overhead of the notification mechanism would be entirely worth it.

5.3.5 *LPC* Batch-Insert Approach Validation

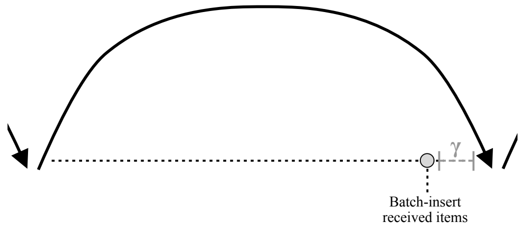
As previously described in Section 4.9.3, we implemented a periodic batch-insert mechanism to store items in the Thyme-Infra's *Local Popularity Cache*. Furthermore, we argued that batch-inserting items, at multiple points, during each popularity decision cycle would provide the best tradeoffs in terms of both system performance — since the *AMC* algorithm would only be executed during a handful of times, per cycle, with multiple objects in each call — as well as user experience, since items would gradually be available to be downloaded as each batch-insert is performed.

To prove our assertions, we solely focused on validating this feature with the execution of this evaluation scenario. Consequently, we executed the trace previously specified, using a single Thyme-Infra node, and kept track of the total time the CPU spent processing *Local Popularity Cache's AMC* as well as the number of download operations, executed by the mobile nodes, that were served by the infrastructure instead of other peers. Each popularity decision cycle was 30 seconds apart.

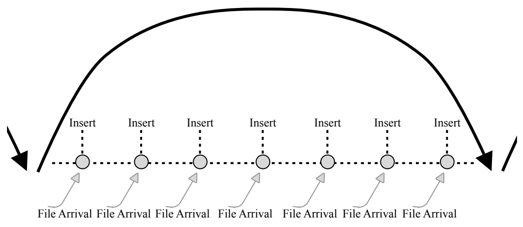
With the goal of simulating the real time that a mobile client would take to reply to a download request from the infrastructure, each virtual node responds within 20 seconds with a 80% probability and within 20 to 30s with a 20% probability. We believe that the 20 seconds represents an acceptable interval for most devices which takes into account the fact the Thyme application could be in the background, and thus be impacted by the Android OS itself which actively throttles the processing rate of background applications. With the extended 10s we are considering older mobile devices or devices that simply froze due to other active high-intensive tasks and therefore are unable to reply earlier.

The evaluation scenarios are as follows:

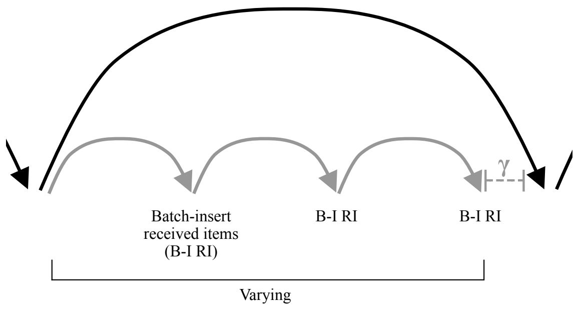
- Bath-inserting all received items right before the next execution of the Popular Decision cycle (Fig. 5.9a), which will be referred as "BIBPDC";



(a) Insert all before the next Popular Decision cycle.



(b) Insert during each individual item arrival.



(c) Periodically batch-insert items between Popular Decision cycles.

Figure 5.9: Evaluation Scenarios for the "Validation for our Batch-Insert Approach on the *LPC* Cache".

- Inserting individual items into the cache as soon as they arrive at the infrastructure (Fig. 5.9b);
- Batch-inserting items that arrived at the infrastructure, at multiple points, during each popularity decision cycle. This specific approach will be further evaluated by varying the number of batches executed (Fig. 5.9c).

Regarding the "Insert at Arrival" approach, we acknowledged from our initial tests with multiple worlds that while the *Local Popularity Cache* was not full the *AMC* would essentially fallback to a one dimensional cache design approach, where each put operation would always increment its world's cache container size by one. And while we verified that until the moment the cache was full the number of items served by the infrastructure was higher than the other methods, as they were more readily available to the users, each insert was blindly competing with users from other worlds for storage space. However, when the cache reached the maximum capacity specified by the developer, the *AMC* equation was unable to change (increase or decrease) the containers' size from that point on due to the one-by-one nature of inserts with this approach, which have minimal impact on the outcome of the formula. Therefore, if we consider a scenario where the world 1, 2 and 3 inner caches occupy 60%, 30% and 10% of the main cache, respectively, right when it becomes full, that particular layout will be unaltered throughout the entire duration of the event even if the world 1 population drops and the popularity of the rest of the worlds

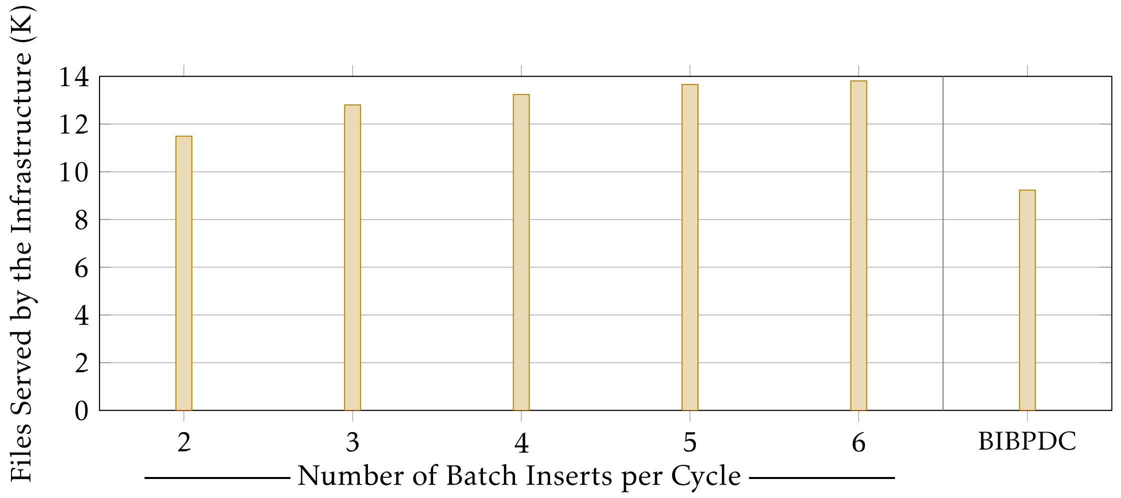


Figure 5.10: Total number of items served by the infrastructure depending on the batch insert method utilized by Thyme-Infra.

greatly increase. Thus, due to the reasons mentioned above, we discarded the "Insert at Arrival" scenario as it is an unfeasible approach.

Fig. 5.10 exhibits the impact on the number of files that were served by the infrastructure during download requests from the mobile nodes when using both our *Batch-Insert* and BIBPDC approaches. We decided to limit the number of batch inserts to 6 as an increased number would tend to be similar to the "Insert at Arrival" mechanism. At a first glance, we verify that utilizing the BIBPDC provides a worse performance in regard to the studied metric, resulting in the infrastructure to serve approximately 9000 files to the clients. This outcome was predicted as the cache contents are only updated every 30 seconds. On the other hand, just by changing the approach to the minimum batch-inserts possible (i.e. two) it is possible to determine an increase of ~2500 files sent back to the downloading clients, which translates to a roughly 25% improvement. From that point on, the number of files served by the infrastructure layer will gradually increase at a slower rate until the 6 batch-inserts are reached: going from ~11500 files in the 2-batch approach to approximately 14000 with 6 batches. Therefore, from this results, we can affirm that our insertion mechanism always provides better results in terms of files served than inserting items at the end of each cycle.

However, when we observe the total CPU time to process the re-execution of the *AMC* algorithm during both of these approaches, in Fig. 5.11, the increased availability provided by our solution is correlated with an higher processing overhead, as expected, that grows exponentially with the amount of batches. That is, in the BIBPDC approach, the *Adaptive Multipart Caching* mechanism represented approximately 496 ms in processing overhead, while batch-inserting 2 and 6 times per cycle accounted for 515 and 729 ms. Although the difference between all of this workflows are essentially negligible in the grand scheme of things, it is necessary to take into consideration that the machines that

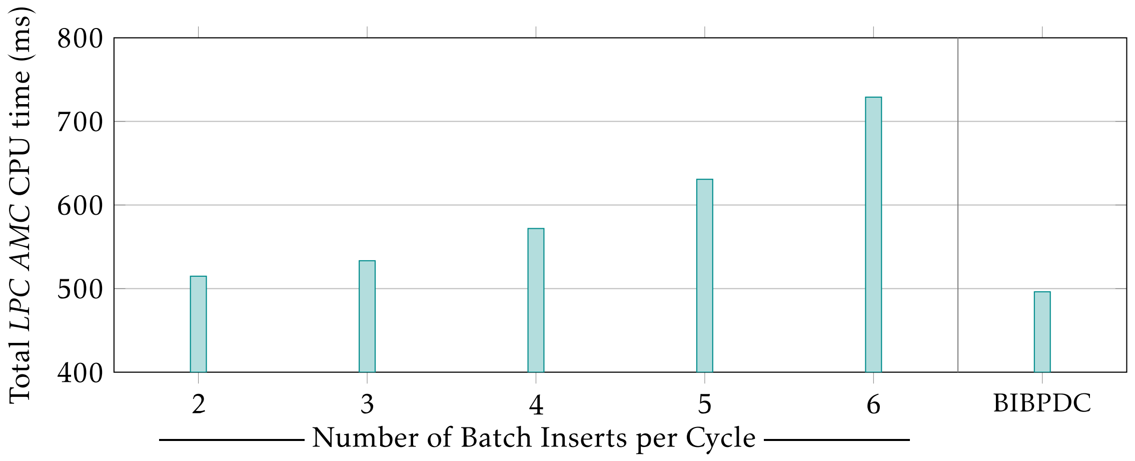


Figure 5.11: Total CPU time spent by the infrastructure processing the *AMC* algorithm depending on the batch insert method utilized by Thyme-Infra.

executed the simulation traces contained server-grade hardware, with Intel® Xeon® CPU E5-2620 v2 @ 2.10GHz and 64 GB of RAM. Therefore, it is expected that the CPU time dissimilarities will be further accentuated with a more limited Base Station hardware.

Nonetheless, we believe that the flexibility provided by our batch-insert mechanism will allow the application and infrastructure developer to fine tune the system to its needs and achieve a better performance. For instance, this can be particularly useful if a considerable amount of files are published every popularity decision cycle, leading to a higher content turnover in the Thyme-Infra caches. Therefore the developer will be able to adjust the properties for this workflow and increase the number of batches per cycle in order to increase the availability of newly created files.

5.3.6 Global Unpublish

Throughout the execution of the previous testing environments it was possible to evaluate the effectiveness of the *Global Unpublish* workflow, due to the fact that every single simulation trace contained multiple unpublish operations, that is: if an unpublish operation executed by a client is not only processed successfully within its mobile peers and by the nearest infrastructure node, but also transmitted to every other infrastructure node in the system and their respective clients. Although we could not find any meaningful metrics to validate this particular property, we verified that in every scenario, particularly with two or more Thyme-Infra nodes running simultaneously, all unpublish operations issued by mobile clients were correctly executed by all infrastructure nodes (i.e. the referenced item was removed from the local caches, if present, and auxiliary data structures) and all nodes, from each *BSS*, residing in the respective indexing cell. Subsequently transmitted subscriptions related to the unpublished item's tag did not receive a reply with that particular item during the notification step. Thus, we can assert that the successful rate of *Global Unpublish* mechanism was 100% in every instance, independently of the number of mobile nodes as well as infrastructure nodes.

5.3.7 Self-established Cell Stable Node Failover

Similarly to the "Global Unpublish" evaluation, during the execution of our tests we also forcefully terminated multiple virtual nodes, throughout the execution of the traces, and confirmed that the process of disseminating each cell's data to the infrastructure was properly carried out despite the cessation of the node that would be self-established as the stablest. Even when we stress tested this functionality by shutting down successive self-established stable nodes for a given cell in consecutive dissemination cycles, the next stable node would always be able to propagate not only the current timestamp's cell data but also all other previous cell information that the infrastructure was not informed of. The only scenario where we expect this entire failover process to stop working as intended is when there are no other mobile nodes within a cell, due to all of them failing or simply leaving the network.

5.4 Comparison with the Previous Version of Thyme

Unlike the "Absolute Metrics Evaluation", this section is comprised of a variety of specific evaluation schemes that will be used to directly compare the execution of the previous version of Thyme, i.e. the geographical implementations without the infrastructure layer, against its latest iteration along with newly developed Thyme-Infra counterpart.

5.4.1 Simulated Evaluation

To perform the simulated evaluation, both versions of Thyme will execute the same trace file (100 nodes) and the produced results will be directly compared. Moreover, for the geographical version, we will establish a 3×3 geographical grid (9 cells) as the layout of the system. Consequently, to allow a balanced comparison, we will also employ a 9 logical cells-approach for Thyme-Infra. In both versions, nodes will be randomly assigned to a cell at runtime. However, regarding the latest iteration, we will utilize a single Thyme-Infra node to aid the system.

Total Mobile Traffic Generated. In this particular evaluation proposal, we aim to compare how much traffic, and its volume in size, is generated by an environment of multiple mobile nodes executing the previous version of Thyme versus the latest iteration along with the inclusion of infrastructure resources. Furthermore, in order to establish a fair comparison between both versions, we did not allow the mobility of nodes throughout the execution of the trace to avoid the additional traffic, as will be explain in the next evaluation scenario. Lastly, we increased the size of published files from 64 bytes to 100 KB and their description (i.e. miniature or thumbnail) to 20 KB in order to get a glimpse of the outcome of the system, on a real-world setting, with more realistic file sizes.

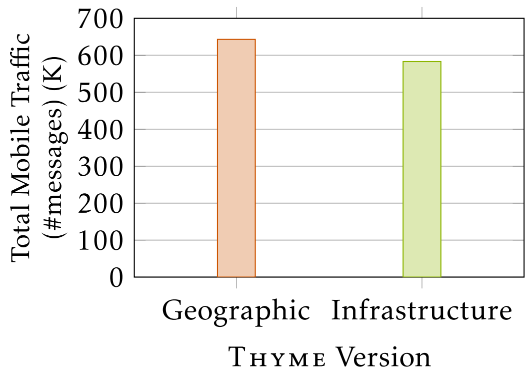


Figure 5.12: Comparison of total messages sent between both versions of Thyme when executing the same trace.

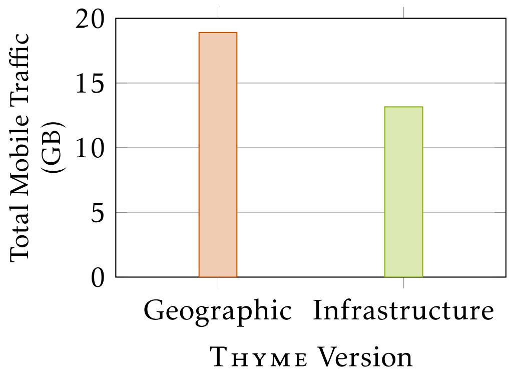


Figure 5.13: Comparison of the generated traffic size between both versions of Thyme when executing the same trace.

Fig. 5.12 displays the total traffic (number of messages) produced by the mobile nodes in each of the scenarios. As expected, the introduction of infrastructure resources and the paradigm change from geographical cells to logical ones results in the reduction of messages between mobile nodes, namely a ~10% reduction of around 60000 messages in this particular trace. This decrement in the traffic from the newest iteration of Thyme plus Thyme-Infra can be attributed to a variety of factors, such as the absence of intermediate routing messages as well as the retrieval of published items directly from the infrastructure.

Similarly, the chart in Fig. 5.13 depicts the comparison between the total size of the generated traffic between both versions of Thyme. While in the geographical version the messages sent accounted for ~19 GB, the infrastructure-aided version generated 13 GB in traffic for the same trace, essentially providing a ~31% reduction in the overall mobile traffic size. The reasons for this discrepancy in the latest iteration are similar to the ones referred in the previous point:

- Fewer files need to traverse through mobile nodes as they can be retrieved directly from the nearest infrastructure node;
- No multi-hopping routing strategy, thus no additional routing messages between cells. This particular detail can have a considerable impact on the size of the traffic: in the previous version, if a node was about to disseminate a file back to the requesting node and they were at least 1 cell apart, the file would essentially be sent in a cell node-by-cell node basis along the routing path in order to reach the destination, creating a lot of undesired heavy traffic;
- Reducing the number of items sent by mobile nodes right after a subscription. While in the Geographical version of Thyme a subscription would lead to a transmission of an initial notification back to the subscriber containing all relevant metadata objects of files published so far, that same notification in our solution would contain

only the relevant items for the subscribed tag that are currently not stored in the nearest infrastructure node.

Ultimately, we are able to conclude that, regarding the system's mobile traffic aspect, the new version of Thyme along with Thyme-Infra provides better results across the board, presenting a lower traffic as well as traffic size. Thus, it is possible to correlate this findings with an inferior computing and battery utilization by the mobile nodes.

Mobility Overhead. While the previous version of Thyme allows full mobility of its clients without compromising the soundness of the system, in order to support it the system makes use of geographically bounded cells and the respective *GHT* which directly imposes some tradeoffs in terms of performance. That is, when a mobile node moves from one cell to another it initiates a housekeeping process that essentially updates every piece of information, distributed throughout the system, that contains the previous node address to reflect the transition to the new cell.

For instance, considering the described 3×3 grid layout, if a client currently has at least one active subscription stored in each of the 9 cells and is a passive replica of at least one file indexed by each of the cells (which we believe both aspects to be very reasonable), and moves from one place to another, thus transitioning to a new cell, the following traffic is generated by the system (being $\#nodesPerCell$ the number of mobile nodes within each of the 9 available cells):

- The transitioning mobile node first notifies all of its original cell neighbors that it is currently in motion and that it wishes to be excluded from contributing to it while moving, sending $\#nodesPerCell - 1$ messages to achieve this;
- When the node stops it will then notify all the 9 cells to update its subscriptions with the new address in order to continue receiving notifications when a relevant item is published. Therefore, $\#nodesPerCell \times 9$ messages are sent during this step;
- On top of that, the mobile node will alert each cell in order to update its currently replicated items' metadata storage locations, sending an additional $\#nodesPerCell \times 9$ messages;
- Then, since the client will become an active participant in managing and contributing to the new cell, it will send a request to one of its neighbors to receive all the cell's P/S data, such as the active subscriptions indexed by it, totaling in 2 messages transmitted (request and reply).

Fig. 5.14 showcases the growth in the total traffic generated by the system during this particular scenario as the number of nodes located within each cell increases. To facilitate the translation of the previous traffic steps into a graphical format, we ignored all intermediate multi-hopping routing mechanism, along with their respective messages,

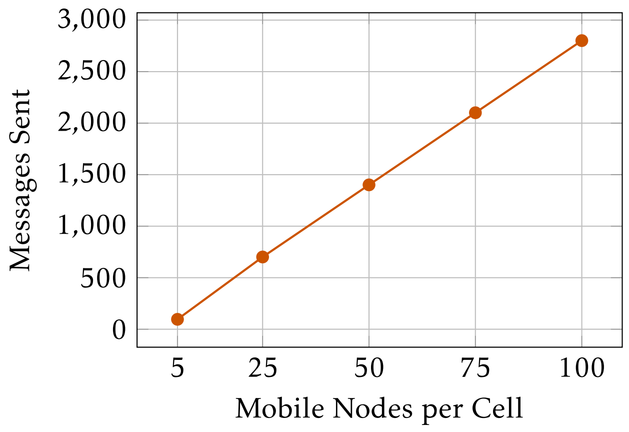


Figure 5.14: Total messages sent by the system (Geographical Thyme) when a single mobile node moves from one cell to another.

as well as we are assuming that each cell has the exact same amount of active mobile clients.

As we can demonstrate from the gathered metrics, the total number of messages disseminated grows linearly along with the number of mobile nodes within each cell. On top of the results displayed in Fig. 5.14, if we were to consider the number of items previously published by the moving node, the generated traffic would increase by $N_{\text{pubs}} \times \#nodesPerCell$ messages (where N_{pubs} represents the number of published items by the client). Ultimately, in this scenario, using the geographical version of Thyme would culminate in the dissemination of at least 96 housekeeping messages, on the lower end of the spectrum, and at least 2801, with 100 clients located within each cell, every single time a mobile node transitions from one cell to another. Moreover, since edge clients are inherently mobile, we argue that this procedure will occur regularly throughout the duration of an event.

When compared to our proposed iteration of Thyme, due to the introduction of infrastructure resources and the logical cells paradigm, there is absolutely no traffic generated during the movement of mobile clients, saving them computing and energy resources. Thus, our version presents itself as the clear winner, especially if we consider events like parties where users are encouraged to be in constant movement.

5.4.2 Real-world Evaluation

Operations Latency. The latency of a system's operations is one of the most critical metrics, especially when that system is built for customer-centric applications. Thus, in this evaluation, we measured the latency of each Thyme P/S operation (Publish, Subscription, Unpublish, Unsubscription and Download) and compared the outcome of every single one using the previous implementation of Thyme against ours in order to quantify the speed improvements with the introduction of infrastructure resources as well as the logic cells paradigm change. To perform this assessment we utilized very small

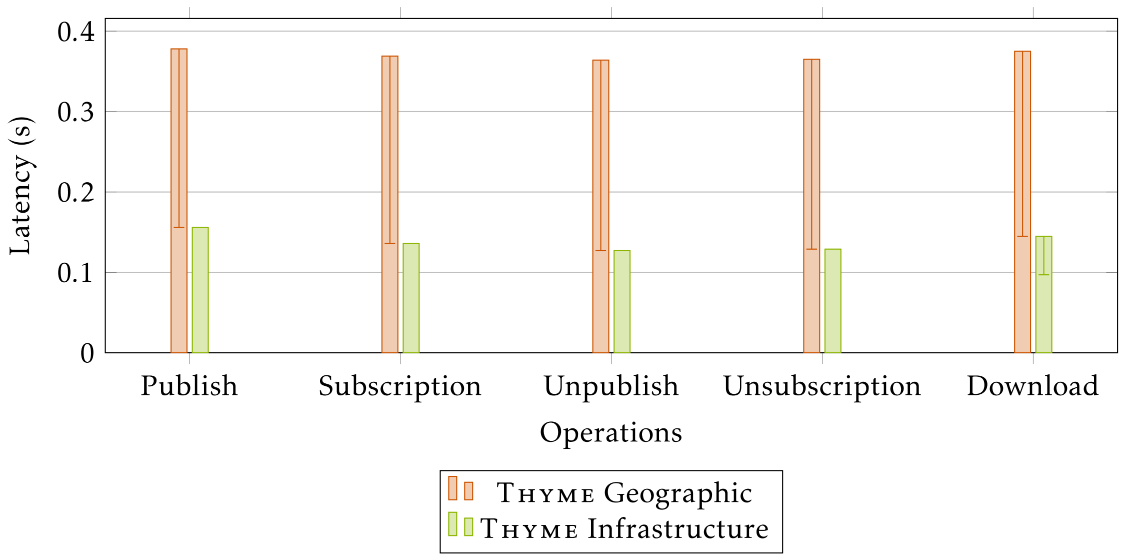


Figure 5.15: Operations latency comparison between both version of Thyme.

photographs — with 64 bytes for both the picture itself and its miniature — to determine the latency baseline without getting affected by an increased file size. Furthermore, we only utilized the Motorola Nexus 6 smartphones to estimate the operations' latency to minimize disparities between hardware readings.

In Fig. 5.15 we showcase the average measured latency of all the mentioned operations. For the geographical version of Thyme, the base values represent the latency when the sender is 2 hops away from the operation receiver, while the negative error bar shows the value when distance between both sender and receiver is just one hop (i.e. neighboring geographical cells). On the other hand, the error bar in Thyme Infrastructure's download operation depicts the latency when the file request is disseminated to the nearest Thyme-Infra node instead of being transmitted to the mobile nodes.

One of the biggest takeaways from this chart surrounds the fact that the utilization of a multi-hopping routing strategy has a considerable impact on the overall latency of an operation, resulting in a 170% increase — on average — in latency. We consider this outcome to be expected as the routing node essentially has to receive and send the message from the origin to the destination as well as send the operation's response from the destination back to the initial sender, essentially culminating into two Rx and two Tx transactions just from the routing node. While the routing procedure is logically non-demanding, its workflow is almost entirely I/O bound which contributes, on a higher degree, to the delay. Although we were not able to test the impact with an increased number of hops, due to shortage of testing hardware, we expect the increased delay to be linear to the number of hops.

However, when eradicate the routing middleman and position both nodes in adjacent cells, the system falls back to a mode similar to our approach in the latest version of Thyme, where peer-to-peer communication channels are established directly. Even with

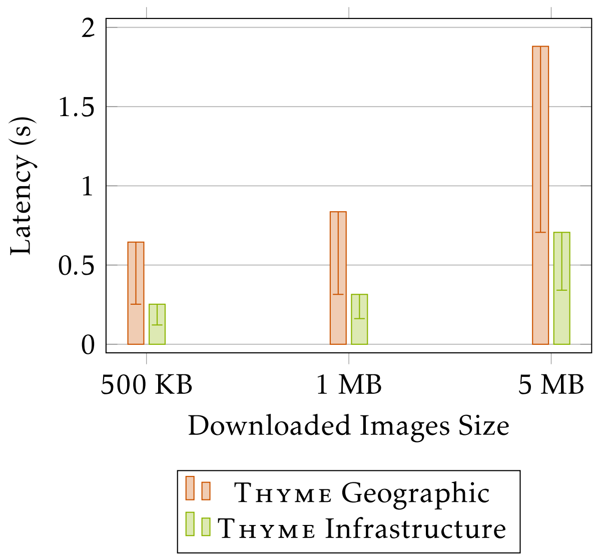


Figure 5.16: Download latency comparison between Thyme Geographic and our iteration of Thyme for multiple file sizes.

the introduction of infrastructure resources and the additional logic that accompanied this alteration, such as the management and gathering of the cell's data to be periodically disseminated to the infrastructure, we have managed to match the latencies to the ones obtained in the best case scenario (1 hop) of the previous version of Thyme. We achieved this by performing any new computations after the operation has been fully executed, thus not impacting the process-and-reply workflow.

Regarding the download operation, this is further improved in our proposal when the request is served directly by the nearest infrastructure, which leads to a ∼33% reduction in the operation's latency.

Hence, we verify that the infrastructure-aided version of Thyme poses itself as a direct improvement, matching the latency of the best case scenario of geographical Thyme for every operation while still providing an additional reduction in the download operation latency by requesting the item from the infrastructure.

Moreover, from the set all of Publish/Subscribe operations in Thyme, the publish and download (retrieval) are the ones that are more susceptible to larger files, as the file miniature and the file itself, respectively, are disseminated during each process, resulting in an increased traffic size and therefore a larger latency. On the other hand, the rest of the operations are set to present essentially the same latency results, as displayed in Fig. 5.15, for every scenario as they are completely independent of the associated item size. Therefore, we focused and further tested the download operation with a more realistic collection of file sizes and displayed the latency results of both version of Thyme in Fig. 5.16, in a similar manner as the previous chart.

As estimated, the download latencies for the 500 KB, 1 and 5 MB files follow approximately the 170% additional delay when comparing the 2-hop routed workflow against the peer-to-peer approach. Similarly, we verify that the latency difference by getting the item from the nearest Thyme-Infra node, across these files, is consistent with the initially recorded 33% decrease. Thus, to download a 5 MB file, a client using the Geographic Thyme will take at least roughly 2 seconds if routing is needed and 0.7 seconds if not, while our version will take at most 0.7 seconds and approximately 0.34 seconds if the item is available in the infrastructure, which we consider to be completely adequate for real-time, interactive applications.

General Battery Consumption. Along with the system's operations latency, the general user battery consumption is also a critical metric that needs to be measured and taken into account. This is particularly important for systems deployed at the edge where not only will the costumer be able to interact with it but will also be to participate and actively contribute its resources to it. Therefore, for this evaluation, we tested the battery footprint of each Thyme client operation (Publish, Subscription, Unpublish, Unsubscription and Download) — in Joules — with the Geographical version and compared the gathered results with our proposed iteration of the system. However, since the system is comprised of a cluster of nodes working together as a way to maintain it, our approach to measuring battery lies on a cumulative foundation, that is, during the execution of a given operation we tracked not only the sender's battery overhead but also every single node's that were essential to the transmission and processing of that operation.

To perform the battery measurements we utilized a custom module, developed in-house, that periodically polls the information from the Android's BatteryManager class [71].

Similar to the reasons discussed in the latency evaluation, we also utilized 64-byte photographs and miniatures and a single Thyme-Infra node to aid the latest version of Thyme. Moreover, we chose to keep executing this tests only on the Motorola Nexus 6 smartphones to reduce the hardware variance.

Fig. 5.17 shows the average battery utilization for all the mentioned operations using the previous version of Thyme, from the standpoint of the sender, the singular router and finally the recipient that processes the request and sends the response back to the client. From this chart we verify that all of the operations carry an overall very similar cumulative battery footprint, close to 2 Joules. Individually, the sender uses an average of 0.55 J while the recipient consumes ~0.51 J. On the other hand, the router utilizes approximately 0.89 J, which makes up the bulk of the entire operation's cumulative battery consumption, at 46%. This was also expected due to the same arguments presented in the latency evaluation section. On top of that, the routing battery usage component in the graph is foreseen to grow linearly with the amount of intermediate hops as well.

Alternatively, Fig. 5.18 presents the same operation-by-operation battery consumption evaluation for the infrastructure-aided version of Thyme. The first thing that we are able to assess is that, due to the absence of the routing mechanism, the cumulative metric is only comprised of the sender and the recipient actors. Consequently, if at least one routing hop is needed by the Geographical Thyme to communicate with the sender,

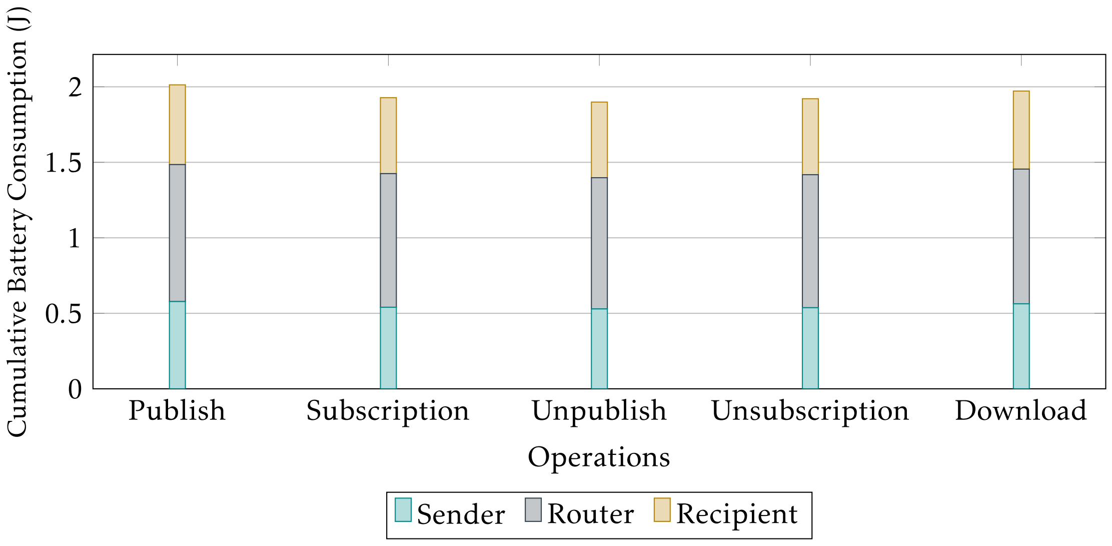


Figure 5.17: Cumulative battery consumption during each Publish/Subscribe operation in the previous iteration of Thyme.

our solution presents a much lower battery utilization aggregate for every operation (with 37% improvement on average), ranging from approximately 1.1 J with the Unsubscription procedure to ~1.4 J while subscribing. The lower end is improved even further when a download is successfully processed by the infrastructure, thus eliminating the recipient from the equation and becoming the least resource-intensive operation, using a total of 0.5 J. On the other hand, the downside of including the infrastructure resources is most noticeable on the Subscription and Unpublish operations as they were altered to simultaneously transmit the message to the nearest infrastructure, leading to a battery consumption increase of approximately 61% on the sender node when compared to the previous iteration.

Regarding the Unpublish operation, while it is indeed worse in terms of energy utilization for the client's smartphone that initiates the process, we consider that the dissemination of this message will be highly uncommon, therefore the additional impact will be essentially nonexistent. The same thing cannot be said for the Subscription. However, our implementation for that operation is able to save battery at a much larger scale on the recipient side, as will be described in the "Battery Consumption Analysis on Over-utilized Cell Nodes" testing scenario.

On the recipient side, our system also imposes a bit of a drawback in terms of battery usage during a Subscription and an Unsubscription, namely an increase of 0.034 J. This is caused because these operations affect the cell's data, which forces the recipient node to update its cell data information, resulting in the additional overhead.

Nonetheless, the presented Joules are incredibly small to paint the full picture and give a sense of a real-world scenario and utilization. Thus, we calculated how many times each operation would have to be executed, by each of the system actors, in order to drain 1% of the Motorola Nexus 6's battery (equivalent to 440 Joules) and showcased the results

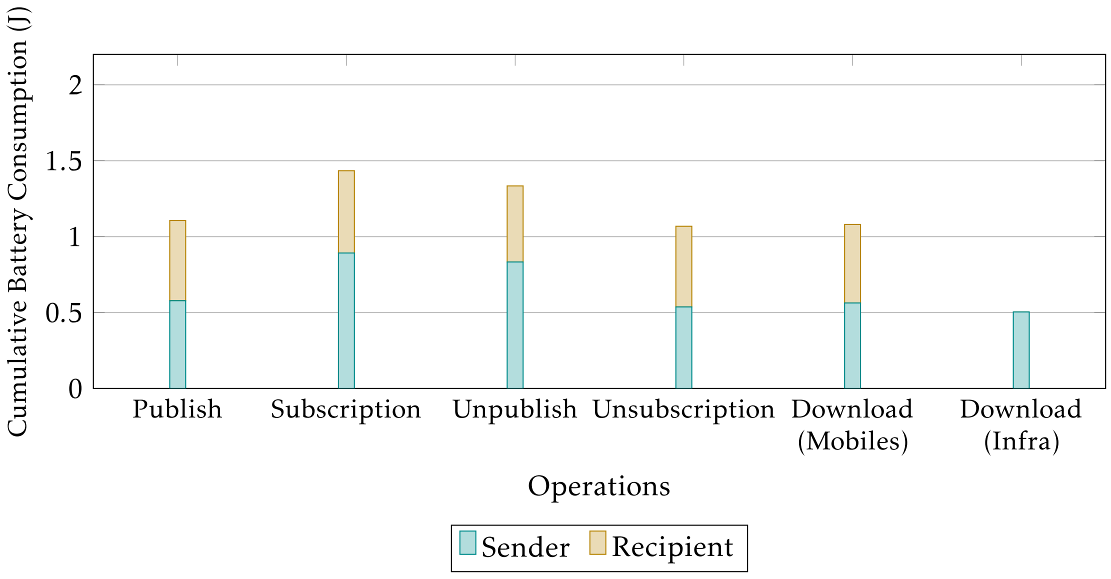


Figure 5.18: Cumulative battery consumption during each Publish/Subscribe operation in our proposed iteration of THYME.

in Table 5.3. As we can verify, there is a direct impact which is bound to happen when the system is altered to also take into account the infrastructure layer as part of the operations workflow. However, we believe that the gathered results exhibit an acceptable view of the system's performance, proving that our version of THYME can be used throughout small to medium duration events without presenting an excessive energy consumption to the user's smartphone.

Table 5.3: Number of operations required to be executed in order to reach a 1% battery consumption (the higher the better).

		Publish	**Subscription**	**Unpublish**	**Unsubscription**	**Download**
Sender	THYME Geo.	761	815	832	819	782
	THYME Infra.	761	493	528	819	782 / 873*
Router	THYME Geo.	481	497	506	499	493
	THYME Infra.	—	—	—	—	—
Recipient	THYME Geo.	833	875	878	875	851
	THYME Infra.	833	812	878	829	851 / —*

* Left indicates if the download was sent to the mobile nodes, while the right value indicates if it was sent to the infrastructure node.

On the other side of the spectrum, Fig. 5.19 illustrates the battery consumption of some operations that are specific to the latest version of THYME, namely the process of finding the nearest infrastructure node(s) through multicast probing and the process where each mobile node checks whether it should self-establish as the cell's stable node, during every cell data dissemination cycle, and actually disseminating that information to the infrastructure (only for the self-established stable nodes).

From this chart one can identify that the "Infrastructure Probe" workflow is the system operation that requires the largest amount of battery power to execute, both from

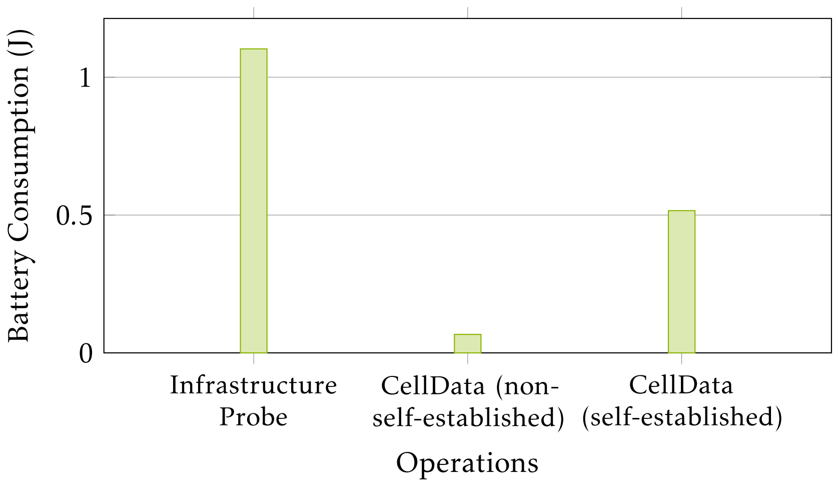


Figure 5.19: Battery consumption of Thyme operations specific to the addition of Thyme-Infra nodes.

a geographical and infrastructure-aided Thyme standpoint, from a single actor. This outcome is completely expected as the application needs to explicitly notify the Android Operating System to allow the device's Wi-Fi stack to process packets not explicitly addressed to the mobile device (i.e. multicast messages, which are automatically filtered as the default behavior), which can cause a noticeable battery drain, in order to probe for Thyme-Infra nodes [72]. Yet, since this operation is only executed once when a mobile node is joining the network, we consider its impact to be negligible and essentially nonexistent.

Concerning the cell data operations, the vast majority of the nodes within the system will simply execute the stable checkup process during each dissemination cycle while only 1 per cell will go further and disseminate the actual data to the infrastructure. To consume 1% of the device's battery, a non-self-established node would have to process approximately 6500 cycles while a self-establish one would have to disseminate the cell data to the infrastructure 853 times. Thus, we find that the overhead caused by this infrastructure-only additions are completely acceptable due to the minimal battery consumption for the "CellData (non-self-established)" operation and the expected low percentage of nodes that will have to perform the "CellData (self-established)" workflow.

Battery Consumption Analysis on Over-utilized Cell Nodes. One of the disadvantages of Thyme and its respective Publish/Subscribe approach lies on scenarios where only a handful of tags are relevant and used/subscribed to. For instance, during a football match in a stadium, users are expected to subscribe and publish most of the data with a focus on tags related to the name of the playing teams as well as the tag "goal". As a consequence of the user behavior in environments like this, the majority of the system operations are disseminated to the cells that are indexing those tags to be processed, resulting in an over-utilization of the nodes residing in those cells, while leaving others

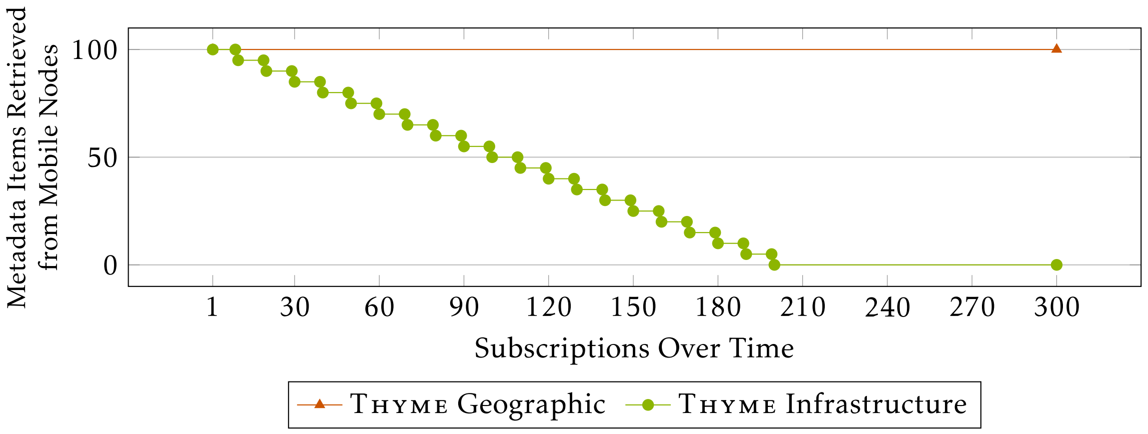


Figure 5.20: Evolution of the number of items retrieved, through mobile means, in each subscription.

and their respective nodes with a much lower workload, which leads to a lower overall performance. This situation can escalate even further if we consider a worst-case scenario where the small set of all the popular tags are indexed by a single cell.

Therefore, in this section, we aim to verify the impact of the newly implemented feature of Thyme where during a subscribe operation the indexing cell node will only return the items that will be not served by the nearest infrastructure node for the same subscription, specifically the battery consumption reduction provided by it when compared to the previous version of the system. To achieve this, we implemented a scenario that would mimic a real-life event at a football stadium: after a team scores a goal, 100 new items with the tag "goal" are published right away and, consequently, users will start to issue subscribe operations for that tag, one in every second, during a total of 300s. The over-utilized cell was composed of 3 Motorola Nexus 6 smartphones while a single Motorola Moto G (2nd gen) was used to periodically disseminate the subscription operations. On top of that, each metadata item contained a miniature of 100 KB and, for the infrastructure-aided Thyme counterpart, we deployed a single Thyme-Infra node that offloaded 5 new items into its caches every 10 seconds

In Fig. 5.20 we can automatically verify the direct consequence of having the infrastructure node on the items retrieved during each subscription operation. That is, when the Thyme-Infra instance repeatedly retrieves 5 items to be stored locally and notifies the nodes within the item's indexed cell, the original mobile node using Thyme + Infrastructure starts to receive less and less items through mobile means as they are increasingly obtained directly from the nearest infrastructure service. On the Geographical Thyme the presented behavior is expected (i.e. the 100 published items are always returned during a subscription) as it does not have any functionality that improves this process.

Consequently, Fig. 5.21 translates the gradual reduction in mobile traffic, with the addition of the infrastructure resources, into the total battery utilization throughout the

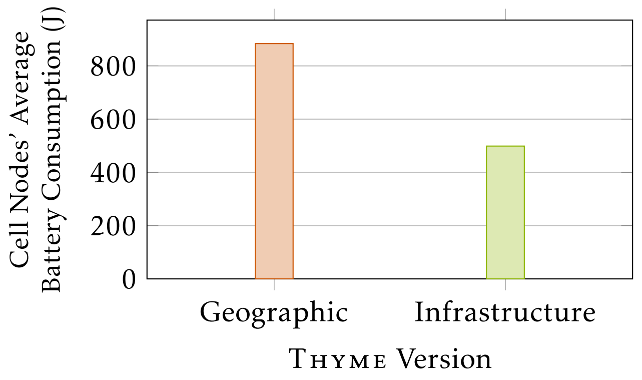


Figure 5.21: Average battery consumption on the over-utilized cell nodes during the subscription scenario.

execution of this scenario. Using the Geographical version of Thyme, each mobile node within tag "goals"'s indexing cell used, on average, ~884 J while the latest iteration only utilized approximately 499 Joules, meaning that our solution resulted in a 44% reduction in battery consumption. Concretely, each cell node consumed roughly 1% and 2% of the smartphone's battery using Thyme + Thyme-Infra and Geographical Thyme, respectively.

In conclusion, we are able to drastically reduce the mobile nodes battery consumption with the implementation of our subscription workflow. Even though we are trading off a bit of the operation sender's battery life to achieve this, we expect the recipient battery savings to grow linearly along with the total amount of published items that are stored in the infrastructure as well as their miniature sizes (which will make up the vast majority of the subscription's reply message content). On the other hand, the impact on the sender is predicted to essentially stay the same since the messages transmitted by it are completely independent of these variables.

5.5 Comparison with the Cloud Infrastructure

The goal of Thyme and Thyme-Infrastructure, as well as other edge-related systems, is to provide a better user experience to the nearby clients by having higher performance and lower operation latencies when compared to systems deployed in the cloud. Therefore, since we are expecting the download operation to dominate the mobile clients' traffic, we compared our system's latency to a cloud infrastructure when processing downloads from the mobile clients, in order to prove the advantages of using our system instead of relying only on cloud systems.

To set up this evaluation scenario, we utilized Amazon Web Services (AWS) [73] as our cloud infrastructure and stored multiple files, of different sizes, in AWS' S3 service. S3 [74] is an object storage service, essentially acting as a file database, built to store and retrieve any amount of data from anywhere: web sites and mobile apps, corporate applications, and data from IoT sensors or devices. Further, for this testing scenario, we used the S3 service replica located in AWS' London data center.

To evaluate the cloud counterpart we manually downloaded each of the files multiples times, at various parts of the day, and measured how long it took to fully retrieve the item, in seconds, using the *curl* [75] command line networking tool:

```
$ curl -s -w "%{time_total}" -o /dev/null https://s3.eu-west-2.amazonaws.com/thyme.tests/file.dat
```

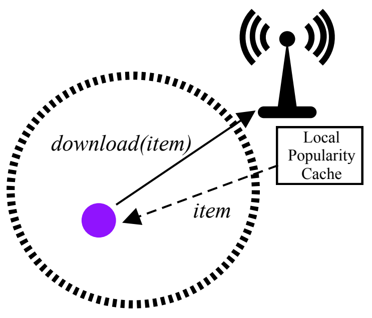
On the other hand, to test our system, we chose to perform the following scenarios, which represent the different possible workflows that can be executed by our system during to process a download operation from a mobile node, and measured the operation end-to-end latency in each of them:

C.1 **Download from nearest mobile replica**: the download request is sent to a neighboring mobile node that is currently replicating the item within the same *BSS*. To enact this scenario, we resorted to using two smartphone devices and one Thyme-Infra node (Fig. 5.22c);

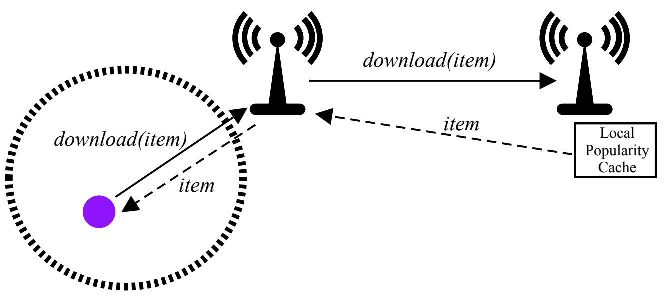
C.2 **Download from nearest infrastructure**: the specified item to be downloaded by a mobile node is served directly by the nearest infrastructure node, from its local storage. To perform this evaluation scenario, we utilized one smartphone for the client and a single Thyme-Infra instance with that item already stored in the *Local Popularity Cache* (Fig. 5.22a).

C.3 **Download from farther infrastructure**: the downloaded item is not currently being replicated by a neighboring mobile node nor it is present in the nearest infrastructure node's storage, but in the *LPC* of the infrastructure node where the item was originally published. Thus, the download request is routed from one infrastructure node to another. To achieve this, we utilized one smartphone device for the requesting client and two Thyme-Infra instances, client's nearest and farthest infrastructure node respectively, with the specified item stored in the second one (Fig. 5.22b).

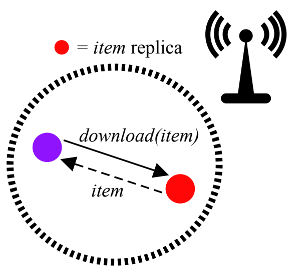
C.4 **Download from farther infrastructure's mobile replica**: lastly, in this evaluation scenario, the downloaded item is neither currently being replicated by a neighboring mobile node, nor it is present in the nearest infrastructure node's storage, nor within the farther infrastructure node where the item was originally published. Therefore, the required object is only replicated by a mobile node in the farther infrastructure node, i.e. the creator of the item itself. To architect this environment, we used two mobile devices — one in each of the *BSS* — where one initiates the download request and the other is simply replicating the item, and two Thyme-Infra nodes (Fig. 5.22d).



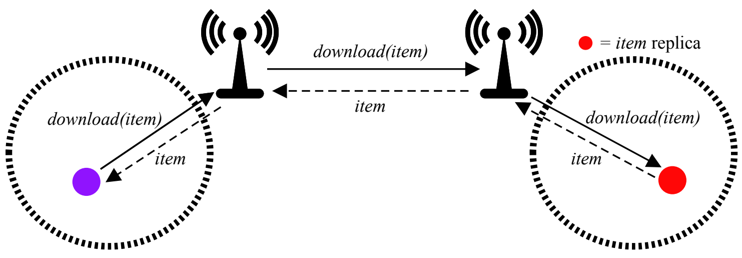
(a) Download from nearest infrastructure.



(b) Download from farther infrastructure.



(c) Download from nearest mobile replica.



(d) Download from farther infrastructure's mobile replica.

Figure 5.22: Evaluation Scenarios for the Thyme + Thyme-Infra part of "Comparison with the Cloud Infrastructure".

In order to establish a fair comparison baseline between our system and the cloud, we have explicitly deactivated the proactive caching mechanism that would cause the item to be cached by the infrastructure node, on scenarios **C.3** and **C.4**, before sending it back to the requesting mobile node. If we kept the proactive caching mechanism enabled, the download latencies after the first execution in each of the mentioned scenarios would essentially be equal to the values obtained in **C.2**. However, by disabling it, each iteration of each scenario will force a complete execution of the download workflow.

The final latency results across all scenarios are visible in Fig. 5.23. With the gathered metrics, it is possible for us to conclude multiple aspects.

First, when only considering the latencies from the Thyme operations, it is possible to verify that retrieving an item from the nearest and the farther infrastructure nodes are the fastest download methods and are relatively close in terms of results across all file sizes. This is expected since the equipments used to act as Thyme-Infra nodes (laptops) contain a more powerful set of Network Interface Controllers with a higher throughput ratio when compared to the ones utilized in smartphone devices. This behavior is also predicted to happen with real Base Station equipments.

Second, apart from the 5 KB file, our system presents itself as a clear winner in terms of latency, providing much lower download speeds across the board than sending the download through the entire network backbone to reach the cloud infrastructure and retrieve the same file. Although Thyme also presents better results for the 5 KB file,

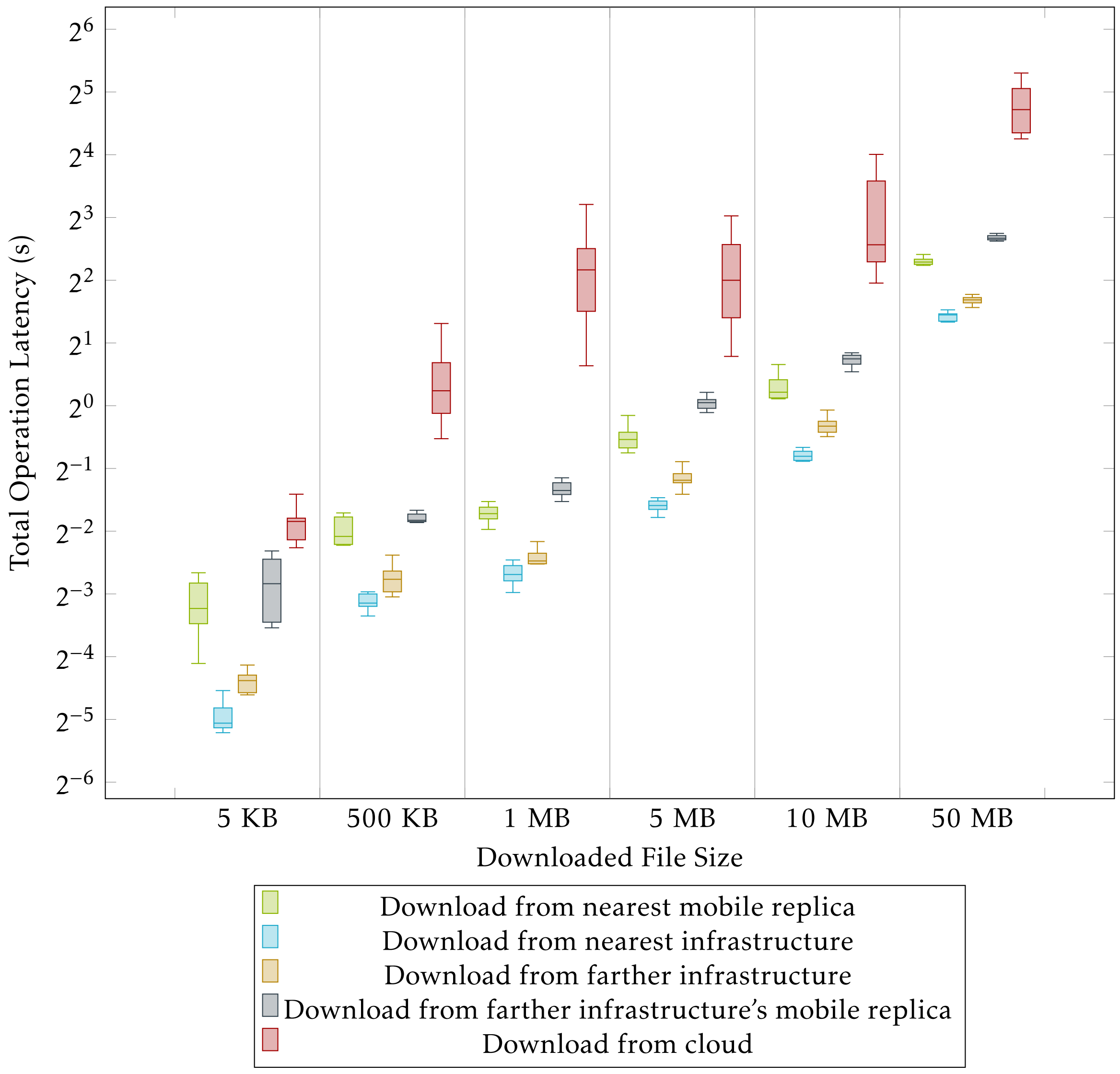


Figure 5.23: Comparison of the download operation latency between multiple files from Thyme + Thyme-Infra and AWS.

the overall difference between our system and the cloud is negligible. Nonetheless, the creation of such small data items in real scenarios will likely be nonexistent if we consider popular data types, such as text documents, pictures and videos, which are usually much larger in size.

Third, if we only consider the 1 MB and 5 MB files downloaded from the cloud, the results obtained through our latency tests are counterintuitive: it was generally faster to download the latter then the former, even though the latter file is five times larger. We believe that this outcome is directly linked to one of the disadvantages of using cloud systems, namely public cloud systems such as AWS: unpredictable traffic at a large scale. Cloud infrastructures effectively represent a communication hub which potentially receives messages from every kind of client and application, causing the requests from all platforms to compete with each other for the servers' processing power and channel

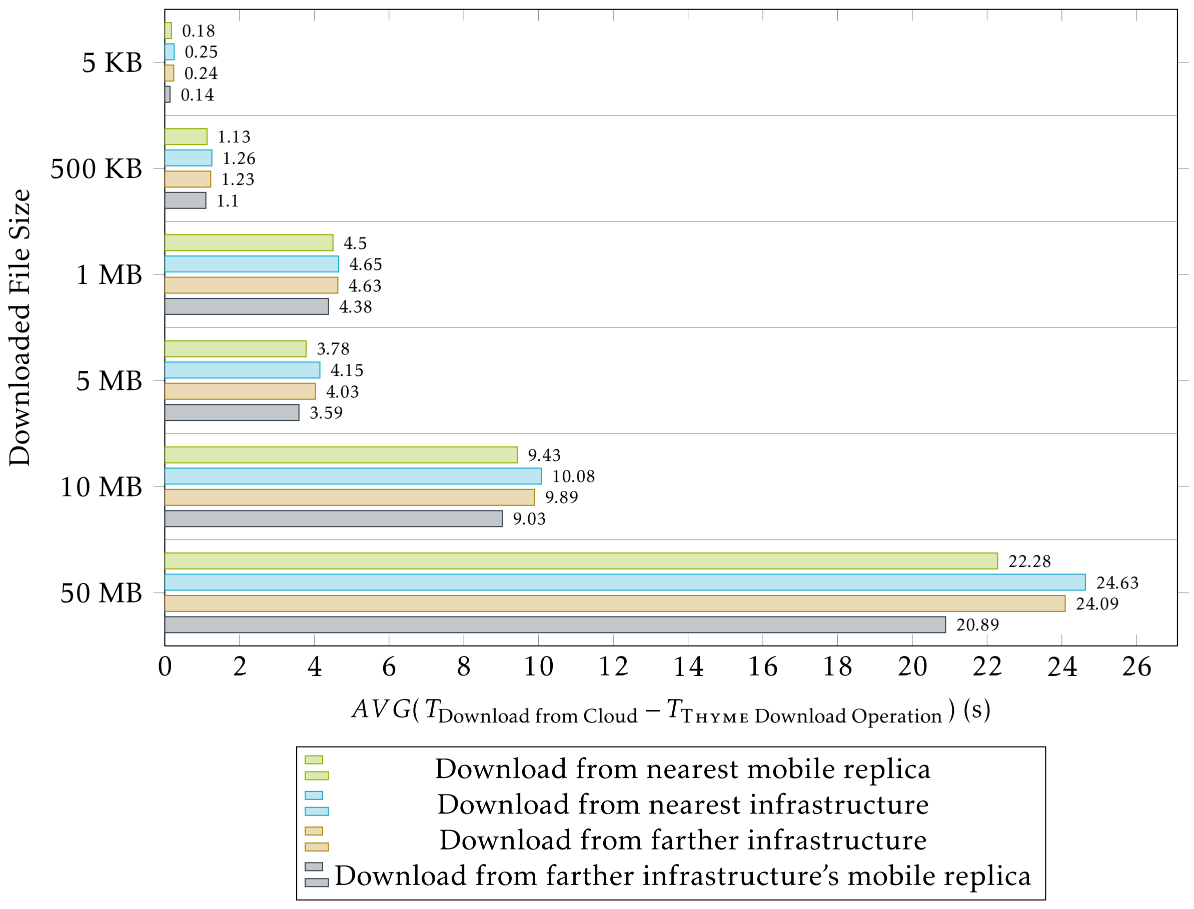


Figure 5.24: Showcase of how much faster, on average, each Thyme download operation is compared to sending the same download request to the cloud.

throughput. Therefore, there is a strong possibility that, during the time-window when we performed the latency evaluation on the 1 MB file, Amazon's London data center was under a higher incoming/outgoing traffic than usual, which directly impacted our results. Consequently, for this reason, we conclude that relying solely on cloud processing for certain latency-sensitive applications is not always adequate.

In Fig. 5.24 we showcase a more streamlined view of the comparison by directly presenting how much faster, in seconds, each Thyme download operation is on average when compared to the cloud counterpart. Ultimately, we conclude that our system is able to provide swifter response times ranging from approximately 4 to 5 seconds faster to ~9 – 10 seconds for the most common file sizes (1 MB through 10 MB) that are expected to be generated and shared via smartphones during events (i.e. selfies, photos and short video clips). This difference is even more accentuated when considering larger files.

CHAPTER 6

Conclusions

6.1 Conclusions

In recent times, we have seen an incredible growth of users adopting mobile devices and wearables, such as smartphones and smartwatches, for communicating with friends, performing day-to-day activities or even work-related procedures. As a consequence of this, the global traffic generated by these non-stationary devices represents the majority of all internet interactions, and is poised to keep growing. While smartphones' hardware capabilities have greatly increased year after year, mobile communications still remain a bottleneck for most applications. We can partially pinpoint this to the companies' cloud infrastructure, which effectively represents a communication hub where all kinds of platforms compete with each other for the servers' processing power and channel throughput. Additionally, wireless technologies used in mobile environments are unreliable, slow and congestion-prone by nature when compared to the wired medium counterpart.

To fix the back-and-forth mobile communication overhead, the "Edge" paradigm has been recently introduced, which aims to bring cloud services closer to the customers, by providing an intermediate layer between the end devices and the actual cloud infrastructure, resulting in faster response times. While initially this paradigm was composed of infrastructure-only solutions, through the deployment of stationary hardware such as gateways and wireless routers, on hand-picked places like factory floors, vehicles and on top of buildings, other iterations of this approach have been researched and proposed by the entire scientific communicate which utilize client's actual mobile devices as the ones managing and coordinating the system itself.

One of these approaches, THYME, employs a novel time-aware Publish/Subscribe scheme, along with a Geographic Hash Table mechanism, and has been proven to be highly effective for data dissemination at edge networks, due to the interactions' loosely coupled nature and scalability.

Although a plethora of solutions already exist on both ends of the edge spectrum, very few actually make use of the combination of both of these actors simultaneously — stationary nodes and mobile devices — to create a truly and fully collaborative storage system.

In this thesis we proposed an edge storage system, built on top of Thyme, that utilizes, when available, previously deployed infrastructure nodes with the goal of joining multiple groups of clients within the same venue, from different locations, and their data into one cohesive and consistent end-to-end storage network. At the same time, the system created between the mobile clients would run independently of the infrastructure and, in the event that the infrastructure node crashes, the P2P would continue to function normally.

Our end goal with bringing both types of actors into a single framework was to address the disadvantages of both parties and offset them by working in harmony. That is, introducing the infrastructure resource — which are inherently more powerful and capable than consumer's devices — allowed us to optimize and offload a portion of the requests and files from the clients into the stationary device, ultimately saving them energy and lowering their processing overhead. Furthermore, since these stationary hardware can be connected through wired means to other infrastructure nodes, we essentially overcame the mobile devices' wireless protocols range limitations and enabled the possibility of distant clusters of users to interact with one another. On the other hand, by having users take an active role on the system, we were able to minimize the occurrences of an overload or a single point of failure from the infrastructure part.

The infrastructure part, named Thyme-Infra, was built on top of two core pillars: a P/S system and a complex caching mechanism. For the first one, we augmented and altered the Publish/Subscribe paradigm proposed by the authors of Thyme to run at the infrastructure-level. Its main purpose was the keep track of the Thyme-Infra node's mobile clients needs and periodically disseminate them to all other nodes in the infrastructure, with the intent of receiving relevant popular items that may have been published in other venue locations and offer them to the local clients. Regarding the second pillar, we developed a novel approach to multi-purpose, multi-application, caches through an adaptive segmentation technique which is based on each cache's clients needs.

On top of all this, during this dissertation, the Thyme framework was also entirely updated to be able to probe and interact with the infrastructure layer.

In our evaluations, our system showed adequate response times for interactive usage, and low energy consumption, allowing the application to be used in a variety of events without excessive battery drainage. When compared to the previous iteration of Thyme, some aspects in Thyme-Infra performed poorly than before due to the supplementary complexity inherent to the introduction of infrastructure resources, such as the additional battery usage on the sender node as consequence of simultaneously transmitting the message to the nearest infrastructure node as well as on the recipient device due to the management of its cell data. However, in the grand scheme of things, our solution was able to match or improve Thyme's operations latency as well as reduce the energy consumption. Additionally, we were able provide a considerably lower overhead to the mobile nodes, in terms of the total mobile traffic generated and its size, which is directly tied to the battery and processing improvements.

We also tested our system's download operation latencies — since we believe the operation is expected to dominate the mobile clients' traffic — against a full-fledged cloud solution deployed in AWS for multiple file sizes and verified that our proposal yielded a considerable speedup across the board.

However, there are inherent drawbacks that should be taken into consideration when utilizing our solution which are absent with cloud solutions. First and foremost the energy consumption of Thyme-Infra nodes during their execution: while we were not able to measure this metric on such stationary devices, there is an obvious electric consumption associated with them that would add up to the cost of maintaining the system. Such consumption analysis and costs would be completely ignored by the developer using a cloud provider. On the other hand, congestion of infrastructure nodes caused by a large client population within a BSS. This particular scenario could indeed be solved by acquiring and deploying more edge hardware while, using the cloud counterpart, the developer could simply configure the provider's load balancing and auto-scaling policies and pay only for the resources used when they are needed. Consequently, with the advantages and disadvantages related to our solution, it is up to the engineer to perform an educated guess on which type of system makes sense and better fits its application needs.

In conclusion, we implemented all the technological aspects that we proposed during the initial phase of this thesis and we consider that this work can be seen has a stepping stone towards a streamlined data dissemination/sharing system for a wide-area setting — like a campus or a football stadium — which utilizes both mobile devices to contribute to the processing, storage and routing needs of the system, as well as infrastructure nodes simultaneously.

6.2 Future Work

Even though we have developed a full-fledged system to what was initially proposed for the context of this dissertation, we believe that, while outside the scope of this project, there are still some areas that could be improved as well as completely new features that could be developed in order to enrich the framework and broaden Thyme and Thyme-Infra applications. Thus, we propose a multitude of aspects to be considered for future work, divided into two main groups: system improvements and novel research opportunities.

6.2.1 System Improvements

Batch-message Communication Layer. In the current form of Thyme-Infra, every time an infrastructure node decides to willingly communicate with a mobile client, it builds the respective message and transmits it right away. At first sight, this behavior should not be considered nor presented as an issue but due to our "stable nodes" approach (Sections 4.5.1 and 4.9.7) we believe that it could be improved. Namely, since in some cases we prioritize the dispatch of messages to a stable node, when the Thyme-Infra instance is unable to communicate with a specific mobile node directly, in order for it to locally route the packet to the desired destination, which leads to scenarios where a stable node is contacted multiple times within a short time-window. Thus, building a communication layer that has a batch-messaging counterpart would be a favorable improvement: before sending the message right away to a client that layer holds the transmission for a short predetermined amount of time and puts it into a batch queue; sequent messages to the same destination would be linked together and, when the dissemination timer fires, Thyme-Infra would combine every single associated message into one packet and finally transmit it. When developed, this batch-messaging approach could also be applied to the infrastructure communication layer.

Although this proposal would increase the overall size of the packets transmitted by the system, the benefits are expected to out-weight this particular outcome since the number of times the mobiles nodes would have to activate their antennas to receive packets would decrease, especially when this antenna's activation intermittence makes up to a considerable amount of battery usage in portable gadgets.

Mobile Pagination Retrieval. One of the most flagrant drawback of the mobile nodes subscription process (Section 4.8.1) and the infrastructure's dissemination of incoming subscription needs reply's data to its clients (Section 4.6, Step 3), lies on the fact that both operations could trigger a dissemination of an extraordinary amount of data back to the user, which would be not only inefficient but also overwhelming. Therefore, adding a pagination layer to each operation's return data that could be controlled by the user, depending on its needs, would potentially contribute to a better system and battery usage performance. To achieve this proposal, each Thyme-Infra instance would have to maintain an additional structure to manage and keep track of its users current page, and the data associated to it, for each of the operations.

Adaptively Tuning the System's Parameters. One feature we believe would result in a better overall performance for Thyme-Infra, independently of the target applications, would be to adaptively fine-tune the system parameters from the specified baseline (Section 4.9.9), such as the number of batch-inserts between popular decision cycles (Section 4.9.3) or the optimal dissemination contract time window (Section 4.6.1), depending on the current active users and their behaviors, or even implement a totally autonomous solution which adapts the system's variables on-the-fly without the need for a configurations file.

Fallback to a Geographic Mode on Infrastructure Failure. In the current state of our solution, while mobile nodes will not crash due to a failure on the nearest infrastructure during the execution of P/S operations, they will essentially be disconnected to the entire system, including neighboring nodes. To overcome this situation, we propose the implementation of a fallback mechanism, in the future, for mobile devices using Thyme to change their communication means to D2D protocols, i.e. Geographical mode and worlds. By doing this, they would be able to continue the storage system between them, even if isolated from the rest of the clients that have their nearest infrastructure nodes running correctly. On top of that, the same mechanism should be able to handle the reconnecting scenario when that same stationary device goes back online.

Allow the Deployment of New Stationary Nodes Midway. As described in Section 4.6.1, each stationary node establishes its own non-overlap dissemination contract against all other infrastructure nodes during startup. However, if the architecture engineer wishes to add new stations to the infrastructure midway through the event in order to accommodate to a larger influx of users, for instance, our solution currently does not provide a "catch-up" mechanism for new actors. Thus, we propose the addition of this procedure to new versions of Thyme-Infra, enabling new infrastructure nodes to be deployed midway by forcing them to notify the current nodes of their presence and initiate the respective transmission of system information, such as the available worlds, along with the recalculation of each non-overlap contract.

Adaptive Multipart Caching. As described in the feature's evaluation scenario, we argue that the *AMC* mechanism could be further improved and refined to function properly under different workloads and population sizes/needs.

6.2.2 Novel Research Opportunities

Data Updates. In their current state, Thyme and Thyme-Infrastructure consider published data as read-only, immutable. Thus, a complete new paradigm change, that we believe would cement the deployment of our proposed systems at the Edge and open up a whole new world of applications that could utilize our frameworks, would be the introduction of data updates to both Thyme, in P2P and infrastructure mode, and Thyme-Infra. Updates would have to be applied locally, within the infrastructure's *BSS*, as well as globally, since the updated item could have been created in another venue's location but it could also have been obtain by users from other Thyme-Infra nodes, to guarantee that all replicas are notified of modifications. Eventually consistent replication mechanisms and data types, such as CRDTs [76] (State, Operation or even Delta-based [77]

[78]), would theoretically be an efficient and performant foundation for implementing this proposal considering the inherent characteristics of Edge networks and their devices.

Inter-Infrastructure Nodes Mobility-Awareness. While briefly touched on in Section 4.4.1, churn continues to play a big role in the development of systems at the Edge, namely Thyme-Infra, even with the introduction of resources from the infrastructure layer. Nonetheless we acknowledge that, in schemes like our own, churn can divided into two distinct categories: a complete and an intra-system churn. In the former declaration, we assume the basic churn's definition as a collective of users that enter and leave the entire system's boundaries, effectively terminating their presence. While in the latter, we consider intra-system churn to be the act of multiple groups of clients leaving one part of the system to join another area, while still connected to it but interacting with distinct (hardware) components. A specific example of this type of churn in the context of Thyme-Infra would be the collective movement of multiple users from their respective *APs* to the coverage of a single Thyme-Infra node, e.g. people moving to the restrooms and restaurants area during a football match half-time break.

Thus, due to the freedom of movement inherent to the users at the edge, we believe that an inter-infrastructure node mobility-awareness mechanism should be researched and ultimately implemented in future versions of our solution to cope with scenarios like this. This research proposal would have to address questions such as:

- How to keep track of the inter-system churn?
- When should one decide to trigger the transmission of data from one infrastructure node to another?
- Which data should be selected to move from one infrastructure node to another with goal of keeping relevant data close to the moving clients?
- How to deal with the eventuality of a system overflow, namely the storage capacity, of one infrastructure node with the incoming data while still keeping the relevant local items safe?

Edge-Tailored Popularity Decaying Function. Every time a Thyme-Infra instance receives a *cell data* message it will update its objects' popularity index by incrementing the number of times it has been downloaded by its clients. As described in Section 4.9.3.1, the infrastructure node uses this value for each object to decide which top items should be offloaded from the users onto the infrastructure itself. However, in its current version, the Thyme-Infra node simply relies on incrementing this index over time, leading to cases where an highly sought-after item during a short amount of time and scarcely accessed from that point on would still be considered vastly popular due to the considerable popularity index value that was gathered initially. Thus, we believe that using a popularity decay function would be a good approach to maintain the items' popularity values fresh and relevant, updating the infrastructure caches' contents as a byproduct of that, or even entirely removing the associated object from the pool of possible popular candidates if the popularity index reaches back to zero. In doing this, the goal is to guarantee that we are using the caches' storage capacity in the most efficient, meaningful way possible.

Consequently, state-of-the-art popularity decaying functions should be researched, such as the Wilson Score [79] which is used by massive systems like Reddit [80], to check whether they still perform efficiently in scenarios at the edge or if a completely novel approach should be proposed and implemented, given the requirements and the behaviors of moving clients in edge networks.

Bibliography

[1] P. Narasimhan and F. Silva. *Hyrax: Crowd-Sourcing mobile devices to develop edge cloud*. 2017. URL: http://hyrax.dcc.fc.up.pt.

[2] *Smartphones worldwide installed base from 2008 to 2017 (in millions)*. Statista. URL: https://www.statista.com/statistics/371889/smartphone-worldwide-installed-base/.

[3] R. Simpson. *Mobile and tablet internet usage exceeds desktop for first time worldwide*. StatCounter. 2016. URL: http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide.

[4] R. Simpson. *India amongst world leaders in use of mobile to surf the internet*. StatCounter. 2017. URL: http://gs.statcounter.com/press/india-amongst-world-leaders-in-use-of-mobile-to-surf-the-internet.

[5] *Cisco Mobile Visual Networking Index (VNI) Forecast Projects 7-Fold Increase in Global Mobile Data Traffic from 2016-2021*. Cisco. 2017. URL: https://newsroom.cisco.com/press-release-content?articleId=1819296.

[6] T. H. Luan, L. Gao, Z. Li, Y. Xiang, and L. Sun. "Fog Computing: Focusing on Mobile Users at the Edge." In: *CoRR* abs/1502.01815 (2015). URL: http://arxiv.org/abs/1502.01815.

[7] Cisco. *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are*. 2015.

[8] I. Stojmenovic and S. Wen. "The Fog Computing Paradigm: Scenarios and Security Issues." In: *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7-10, 2014*. 2014, pp. 1–8. DOI: 10.15439/2014F503. URL: https://doi.org/10.15439/2014F503.

[9] K. Dolui and S. K. Datta. "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing." In: *Global Internet of Things Summit, GIoTS 2017, Geneva, Switzerland, June 6-9, 2017*. 2017, pp. 1–6. DOI: 10.1109/GIOTS.2017.8016213. URL: https://doi.org/10.1109/GIOTS.2017.8016213.

[10] J. A. Silva, R. Monteiro, H. Paulino, and J. M. Lourenço. "Ephemeral Data Storage for Networks of Hand-Held Devices." In: *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*. 2016, pp. 1106–1113. DOI: 10.1109/TrustCom.2016.0182. URL: https://doi.org/10.1109/TrustCom.2016.0182.

[11] S. Kim. "5G Network Communication, Caching, and Computing Algorithms Based on the Two-Tier Game Model." In: *ETRI Journal* 40.1 (), pp. 61–71. DOI: 10.4218/etrij.2017-0023. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.4218/etrij.2017-0023. URL: https://onlinelibrary.wiley.com/doi/abs/10.4218/etrij.2017-0023.

[12] W. Fan, Y. Liu, B. Tang, F. Wu, and H. Zhang. "TerminalBooster: Collaborative Computation Offloading and Data Caching via Smart Basestations." In: *IEEE Wireless Commun. Letters* 5.6 (2016), pp. 612–615. DOI: 10.1109/LWC.2016.2605694. URL: https://doi.org/10.1109/LWC.2016.2605694.

[13] Dropbox. URL: https://www.dropbox.com/.

[14] Google Drive. URL: https://www.google.com/drive/.

[15] Instagram. URL: http://instagram.com/.

[16] F. Cerqueira, J. a. A. Silva, J. a. M. Lourenço, and H. Paulino. "Towards a Persistent Publish/Subscribe System for Networks of Mobile Devices." In: *Proceedings of the 2Nd Workshop on Middleware for Edge Clouds & Cloudlets*. MECC '17. Las Vegas, Nevada: ACM, 2017, 2:1–2:6. ISBN: 978-1-4503-5171-3. DOI: 10.1145/3152360.3152362. URL: http://doi.acm.org/10.1145/3152360.3152362.

[17] J. A. Silva, H. Paulino, J. M. Lourenço, J. Leitão, and N. Preguiça. "Time-Aware Publish/Subscribe for Networks of Mobile Devices." In: *ArXiv e-prints* (Dec. 2018). arXiv: 1801.00297 [cs.DC].

[18] F. Cerqueira. "Um Sistema Publicador/Subscritor com Persistência de Dados para Redes de Dispositivos Móveis." Master's thesis. FCT NOVA, Sept. 2017. URL: http://hdl.handle.net/10362/28553.

[19] *Android*. Google. URL: https://www.android.com.

[20] Z. Wang, L. Sun, M. Zhang, H. Pang, E. Tian, and W. Zhu. "Propagation- and Mobility-Aware D2D Social Content Replication." In: *IEEE Trans. Mob. Comput.* 16.4 (2017), pp. 1107–1120. DOI: 10.1109/TMC.2016.2582159. URL: https://doi.org/10.1109/TMC.2016.2582159.

[21] F. Zhang, C. Xu, Y. Zhang, K. K. Ramakrishnan, S. Mukherjee, R. D. Yates, and T. D. Nguyen. "EdgeBuffer: Caching and prefetching content at the edge in the MobilityFirst future Internet architecture." In: *16th IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks, WoWMoM 2015, Boston, MA, USA, June 14-17, 2015*. 2015, pp. 1–9. DOI: 10.1109/WoWMoM.2015.7158137. URL: https://doi.org/10.1109/WoWMoM.2015.7158137.

[22] R. Wang, J. Zhang, S. Song, and K. B. Letaief. "Mobility-Aware Caching in D2D Networks." In: *IEEE Trans. Wireless Communications* 16.8 (2017), pp. 5001–5015. DOI: 10.1109/TWC.2017.2705038. URL: https://doi.org/10.1109/TWC.2017.2705038.

[23] X. Song, Y. Huang, Q. Zhou, F. Ye, Y. Yang, and X. Li. "Content Centric Peer Data Sharing in Pervasive Edge Computing Environments." In: *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*. 2017, pp. 287–297. DOI: 10.1109/ICDCS.2017.26. URL: https://doi.org/10.1109/ICDCS.2017.26.

[24] A. Jonathan, M. Ryden, K. Oh, A. Chandra, and J. B. Weissman. "Nebula: Distributed Edge Cloud for Data Intensive Computing." In: *IEEE Trans. Parallel Distrib. Syst.* 28.11 (2017), pp. 3229–3242. DOI: 10.1109/TPDS.2017.2717883. URL: https://doi.org/10.1109/TPDS.2017.2717883.

[25] J. Albadarneh, Y. Jararweh, M. Al-Ayyoub, M. Al-Smadi, and R. Fontes. "Software Defined Storage for cooperative Mobile Edge Computing systems." In: *2017 Fourth International Conference on Software Defined Systems, SDS 2017, Valencia, Spain, May 8-11, 2017*. 2017, pp. 174–179. DOI: 10.1109/SDS.2017.7939160. URL: https://doi.org/10.1109/SDS.2017.7939160.

[26] G. Wu, J. Chen, W. Bao, X. Zhu, W. Xiao, J. Wang, and L. Liu. "MECCAS: Collaborative Storage Algorithm Based on Alternating Direction Method of Multipliers on Mobile Edge Cloud." In: *IEEE International Conference on Edge Computing, EDGE 2017, Honolulu, HI, USA, June 25-30, 2017*. 2017, pp. 40–46. DOI: 10.1109/IEEE.EDGE.2017.14. URL: https://doi.org/10.1109/IEEE.EDGE.2017.14.

[27] C. Liu and C. Lai. "A heuristic data update mechanism in unstructured mobile P2P systems." In: *Ad Hoc Networks* 58 (2017), pp. 138–149. DOI: 10.1016/j.adhoc.2016.09.011. URL: https://doi.org/10.1016/j.adhoc.2016.09.011.

[28] E. Skjervold and M. Skjegstad. "REAP: Delta Compression for Publish/Subscribe Web Services in MANETs." In: *32th IEEE Military Communications Conference, MILCOM 2013, San Diego, CA, USA, November 18-20, 2013*. 2013, pp. 1488–1496. DOI: 10.1109/MILCOM.2013.251. URL: https://doi.org/10.1109/MILCOM.2013.251.

[29] X. Wang, J. Ren, T. Tong, R. Dai, S. Xu, and S. Wang. "Towards Efficient and Lightweight Collaborative In-Network Caching for Content Centric Networks." In: *2016 IEEE Global Communications Conference, GLOBECOM 2016, Washington, DC, USA, December 4-8, 2016*. 2016, pp. 1–7. DOI: 10.1109/GLOCOM.2016.7842342. URL: https://doi.org/10.1109/GLOCOM.2016.7842342.

[30] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili. "Collaborative Mobile Edge Computing in 5G Networks: New Paradigms, Scenarios, and Challenges." In: *IEEE Communications Magazine* 55.4 (2017), pp. 54–61. DOI: 10.1109/MCOM.2017.1600863. URL: https://doi.org/10.1109/MCOM.2017.1600863.

[31] A. Elgazar, K. A. Harras, M. Aazam, and A. Mtibaa. "Towards Intelligent Edge Storage Management: Determining and Predicting Mobile File Popularity." In: *6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, MobileCloud 2018, Bamberg, Germany, March 26-29, 2018*. 2018, pp. 23–28. DOI: 10.1109/MobileCloud.2018.00012. URL: https://doi.org/10.1109/MobileCloud.2018.00012.

[32] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. Briggs, and R. Braynard. "Networking named content." In: *Commun. ACM* 55.1 (2012), pp. 117–124. DOI: 10.1145/2063176.2063204. URL: http://doi.acm.org/10.1145/2063176.2063204.

[33] J. Dilley, B. M. Maggs, J. Parikh, H. Prokop, R. K. Sitaraman, and W. E. Weihl. "Globally Distributed Content Delivery." In: *IEEE Internet Computing* 6.5 (2002), pp. 50–58. DOI: 10.1109/MIC.2002.1036038. URL: https://doi.org/10.1109/MIC.2002.1036038.

[34] J. loup Gailly and M. Adler. *zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library*. URL: https://zlib.net.

[35] J. Rodrigues, E. R. B. Marques, L. M. B. Lopes, and F. M. A. Silva. "Towards a middleware for mobile edge-cloud applications." In: *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets, MECC@Middleware 2017, Las Vegas, NV, USA, December 11 - 15, 2017*. 2017, 1:1–1:6. DOI: 10.1145/3152360.3152361. URL: http://doi.acm.org/10.1145/3152360.3152361.

[36] P. J. Leach, M. Mealling, and R. Salz. "A universally unique identifier (uuid) urn namespace." In: (2005).

[37] Google. *Protocol Buffers*. 2017. URL: https://developers.google.com/protocol-buffers/.

[38] Oracle. *Serializable Objects*. URL: https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html.

[39] Oracle. *Package java.util.concurrent.atomic*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html.

[40] Oracle. *Class AtomicBoolean*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html.

[41] Oracle. *Class AtomicInteger*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html.

[42] Oracle. *Class AtomicLong*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html.

[43] Oracle. *Package java.util.concurrent*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html.

[44] Oracle. *Class ConcurrentHashMap<K,V>*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html.

[45] Oracle. *Class ConcurrentLinkedQueue<E>*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html.

[46] Oracle. *Class ConcurrentHashMap<K,V>'s compute*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#compute-K-java.util.function.BiFunction-.

[47] Oracle. *Class ConcurrentHashMap<K,V>'s computeIfAbsent*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#computeIfAbsent-K-java.util.function.Function-.

[48] Oracle. *Class ConcurrentHashMap<K,V>'s putIfAbsent*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#putIfAbsent-K-V-.

[49] Oracle. *Class ThreadPoolExecutor*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html.

[50] B. Manes. *Caffeine*. URL: https://github.com/ben-manes/caffeine.

[51] T. Johnson and D. E. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm." In: *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. 1994, pp. 439–450. URL: http://www.vldb.org/conf/1994/P439.PDF.

[52] G. Einziger, R. Friedman, and B. Manes. "TinyLFU: A Highly Efficient Cache Admission Policy." In: *TOS* 13.4 (2017), 35:1–35:31. DOI: 10.1145/3149371. URL: http://doi.acm.org/10.1145/3149371.

[53] S. Cohen and Y. Matias. "Spectral Bloom Filters." In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. 2003, pp. 241–252. DOI: 10.1145/872757.872787. URL: http://doi.acm.org/10.1145/872757.872787.

[54] G. Cormode and S. M. Muthukrishnan. "Approximating Data with the Count-Min Sketch." In: *IEEE Software* 29.1 (2012), pp. 64–69. DOI: 10.1109/MS.2011.127. URL: https://doi.org/10.1109/MS.2011.127.

[55] R. Karedla, J. S. Love, and B. G. Wherry. "Caching Strategies to Improve Disk System Performance." In: *IEEE Computer* 27.3 (1994), pp. 38–46. DOI: 10.1109/2.268884. URL: https://doi.org/10.1109/2.268884.

[56] Google. *Guava*. URL: https://github.com/google/guava.

[57] S. AG. *Ehcache*. URL: http://www.ehcache.org.

[58] B. Manes. *Caffeine: Benchmarks*. URL: https://github.com/ben-manes/caffeine/wiki/Benchmarks.

[59] H. Scalability. *Design Of A Modern Cache*. URL: http://highscalability.com/blog/2016/1/25/design-of-a-modern-cache.html.

[60] B. Manes. *Caffeine: Memory Overhead*. URL: https://github.com/ben-manes/caffeine/wiki/Memory-overhead.

[61] B. Manes. *Caffeine: Effiency*. URL: https://github.com/ben-manes/caffeine/wiki/Efficiency.

[62] *Java*. Oracle. URL: https://www.java.com.

[63] Intel Corporation. *Intel Atom® C3000-Series Processor Family 5G Base Station*. Tech. rep. 2017. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/atom-c3000-zte-5g-base-station-white-paper.pdf.

[64] ZTE Corporation. URL: https://www.zte.com.cn/global/.

[65] *EchoLife HN8245Q*. Huawei. URL: https://support.huawei.com/enterprise/en/access-network/echolife-hn8245q-pid-21506180?category=product-documentation.

[66] The JUnit Team. *JUnit*. URL: https://junit.org/junit5/.

[67] J. Sedlacek and T. Hurka. *VisualVM: All-in-One Java Troubleshooting Tool*. URL: https://visualvm.github.io.

[68] *Raspberry Pi*. Raspberry Pi Foundation. URL: https://www.raspberrypi.org.

[69] *Arduino - Home*. Arduino. URL: https://www.arduino.cc.

[70] B. Manes. *Caffeine: Expiration for maximumSize*. URL: https://github.com/ben-manes/caffeine/issues/214#issuecomment-353996539.

[71] *Battery Manager | Android Developers*. Android. URL: https://developer.android.com/reference/android/os/BatteryManager.

[72] *WifiManager.MulticastLock*. Android. URL: https://developer.android.com/reference/android/net/wifi/WifiManager.MulticastLock.

[73] Amazon Web Services. URL: http://aws.amazon.com.

[74] *Amazon S3*. Amazon Web Services. URL: https://aws.amazon.com/s3/.

[75] *curl*. URL: https://curl.haxx.se.

[76] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. "Conflict-Free Replicated Data Types." In: *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29. URL: https://doi.org/10.1007/978-3-642-24550-3_29.

[77] P. S. Almeida, A. Shoker, and C. Baquero. “Efficient State-Based CRDTs by Delta-Mutation.” In: *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*. 2015, pp. 62–76. DOI: 10.1007/978-3-319-26850-7_5. URL: https://doi.org/10.1007/978-3-319-26850-7_5.

[78] A. van der Linde, J. Leitão, and N. M. Preguiça. “Δ-CRDTs: making δ-CRDTs delta-based.” In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*. 2016, 12:1–12:4. DOI: 10.1145/2911151.2911163. URL: http://doi.acm.org/10.1145/2911151.2911163.

[79] E. B. Wilson. “Probable Inference, the Law of Succession, and Statistical Inference.” In: *Journal of the American Statistical Association* 22.158 (1927), pp. 209–212. DOI: 10.1080/01621459.1927.10502953. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1927.10502953. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1927.10502953.

[80] Reddit. *Reddit’s new comment sorting system*. URL: https://redditblog.com/2009/10/15/reddits-new-comment-sorting-system/.

2018 A Persistent Publish/Subscribe System for Mobile Edge Computing Pedro Vieira

Pedro Miguel Lima de Jesus Vieira

BsC in Computer Science and Engineering

A Persistent Publish/Subscribe System for Mobile Edge Computing

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

November, 2018

Pedro Vieira

2018

Pedro Miguel Lima de Jesus Vieira

BsC in Computer Science and Engineering

A Persistent Publish/Subscribe System for Mobile Edge Computing

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

November, 2018

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

2018 A Persistent Publish/Subscribe System for Mobile Edge Computing