João André Almeida e Silva

M.Sc. in Computer Engineering

Data Storage and Dissemination in Pervasive Edge Computing Environments

Thesis submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy in
Computer Science

May, 2021

Data Storage and Dissemination in Pervasive Edge Computing Environments

To all those that contributed
in some way to this work.

Acknowledgements

No man is an island. John Donne coined this expression conveying that human beings need to be part of a community in order to thrive. This can very well be applied to the journey that is obtaining a Ph.D. degree. This thesis would not have been possible without the direct or indirect collaboration of a considerable number of people to whom I would like to express my gratitude. First and foremost, I would like to deeply thank my advisers, Hervé Paulino and João Lourenço, for all their support, patience, and guidance throughout all these years. Thank you for putting up with me for so long. I would also like to extend my gratitude to Eduardo Marques and Nuno Preguiça for being part of my Thesis Advisory Committee, giving me very insightful comments that helped this work evolve. Additionally, I want to say a big thank you to João Leitão for all his effort, help, and brainstorming.

I am grateful to the Department of Computer Science of the NOVA School of Science and Technology (FCT NOVA) from NOVA University Lisbon for kindly providing me with several grants during the course of my Ph.D. Also, for the opportunity to teach lab classes of several courses—a passion I did not know I had—, and for being like a second home to me during all these years. I would also like to thank the institutions that provided me with funding so I could pursue my Ph.D. degree. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT-MCTES), through NOVA LINCS (UID/CEC/04516/2013, UID/CEC/04516/2019, and UIDB/04516/2020), grant SFRH/BD/99486/2014, and in the scope of the Hyrax (CMUP-ERI/FIA/0048/2013), DeDuCe (PTDC/CCI-COM/32166/2017), and HiPSTr (PTDC/CCI-COM/32456/2017) research projects.

To my co-workers and friends who frequented the lab of the NOVA LINCS Computer Systems group throughout the years, Albert van der Linde, Bernardo Ferreira, Daniel Porto, David Navalho, João Soares, Pedro Ákos Costa, Pedro Fouto, Ricardo Dias, Tiago Vale, and Valter Sousa, for all the moments of work, fun, and distraction we shared. Also, thanks to all the Master students I worked with during these years, their effort made this thesis a much more robust work, specially to Filipe Cerqueira and Pedro Vieira, with whom I worked more closely. Thank you to several people that I nagged during this work. To Brad Karp, for the brainstorming about GPSR (and GHT) and quick email replies. To Filipe Araújo, for helping me better understand CHR. To João Paiva, for assisting me into fully grasp Rollerchain. And to Mohammad A. Khan, for the long emails helping me understand the full operation of MobiStore. Also, to the guys from the Department of Computer Science (DCC) of the Faculty of Sciences from University of Porto, namely, Eduardo Marques, Fernando Silva, João Rodrigues, Joaquim Silva, Luís Lopes, and Rolando Martins, thank you for welcoming me in DCC, and for all the interesting discussions in many meetings of the Hyrax project.

To Gang do Comes, namely, Catarina Gralha, Diogo Cordeiro, Gabriel Marcondes, Hélder Martins, and Rita Pereira, for all the moments of fun and relaxation, and for always cheering me up. Of course, a very special thanks to Catarina Gralha, for being there for me along the way, for all her patience and affection, for all the adventures we shared, and for being my #1 supporter throughout these years. Thank you for being the rocket to my anchor. To my parents, Glória and Luís, I am heartily thankful for all the opportunities and support over the years. To my sister, Sara, for her support and understanding in my bad humour days. To all the people in my Scout group (CNE Agrupamento 719 Arrentela), specially Hélder Marques, Joana Coelho, Patrícia Correia, Ricardo Garcia, Sara Silva, and Susana Garcia, for their support when I needed the most and for all the moments of fun we shared, I am very thankful. I really miss being around the campfire under a starry sky. Finally, I wish to thank all my family and friends for being part of my life and for their endless support throughout these years.

This document was created using the *NOVAthesis* LaTeX template [159], developed at the Department of Computer Science of FCT NOVA by prof. João Lourenço[1]. Thank you for this magnificent template (it is a real time-saver).

[1]https://docentes.fct.unl.pt/joao-lourenco

“Success is not final. Failure is not fatal. It’s the courage to continue that counts.”
— Winston Churchill

Abstract

Nowadays, smart mobile devices generate huge amounts of data in all sorts of gatherings. Much of that data has localized and ephemeral interest, but can be of great use if shared among co-located devices. However, mobile devices often experience poor connectivity, leading to availability issues if application storage and logic are fully delegated to a remote cloud infrastructure. In turn, the edge computing paradigm pushes computations and storage beyond the data center, closer to end-user devices where data is generated and consumed. Hence, enabling the execution of certain components of edge-enabled systems directly and cooperatively on edge devices.

This thesis focuses on the design and evaluation of resilient and efficient data storage and dissemination solutions for pervasive edge computing environments, operating with or without access to the network infrastructure. In line with this dichotomy, our goal can be divided into two specific scenarios. The first one is related to the absence of network infrastructure and the provision of a transient data storage and dissemination system for networks of co-located mobile devices. The second one relates with the existence of network infrastructure access and the corresponding edge computing capabilities.

First, the thesis presents time-aware reactive storage (TARS), a reactive data storage and dissemination model with intrinsic time-awareness, that exploits synergies between the storage substrate and the publish/subscribe paradigm, and allows queries within a specific time scope. Next, it describes in more detail: i) Thyme, a data storage and dissemination system for wireless edge environments, implementing TARS; ii) Parsley, a flexible and resilient group-based distributed hash table with preemptive peer relocation and a dynamic data sharding mechanism; and iii) Thyme GardenBed, a framework for data storage and dissemination across multi-region edge networks, that makes use of both device-to-device and edge interactions.

The developed solutions present low overheads, while providing adequate response times for interactive usage and low energy consumption, proving to be practical in a variety of situations. They also display good load balancing and fault tolerance properties.

Keywords: distributed data storage, data dissemination, edge computing, publish/subscribe, peer-to-peer, mobile devices, wireless networks

Resumo

Hoje em dia, os dispositivos móveis inteligentes geram grandes quantidades de dados em todos os tipos de aglomerações de pessoas. Muitos desses dados têm interesse localizado e efêmero, mas podem ser de grande utilidade se partilhados entre dispositivos co-localizados. No entanto, os dispositivos móveis muitas vezes experienciam fraca conectividade, levando a problemas de disponibilidade se o armazenamento e a lógica das aplicações forem totalmente delegados numa infraestrutura remota na nuvem. Por sua vez, o paradigma de computação na periferia da rede leva as computações e o armazenamento para além dos centros de dados, para mais perto dos dispositivos dos utilizadores finais onde os dados são gerados e consumidos. Assim, permitindo a execução de certos componentes de sistemas direta e cooperativamente em dispositivos na periferia da rede.

Esta tese foca-se no desenho e avaliação de soluções resilientes e eficientes para armazenamento e disseminação de dados em ambientes pervasivos de computação na periferia da rede, operando com ou sem acesso à infraestrutura de rede. Em linha com esta dicotomia, o nosso objetivo pode ser dividido em dois cenários específicos. O primeiro está relacionado com a ausência de infraestrutura de rede e o fornecimento de um sistema efêmero de armazenamento e disseminação de dados para redes de dispositivos móveis co-localizados. O segundo diz respeito à existência de acesso à infraestrutura de rede e aos recursos de computação na periferia da rede correspondentes.

Primeiramente, a tese apresenta armazenamento reativo ciente do tempo (ARCT), um modelo reativo de armazenamento e disseminação de dados com percepção intrínseca do tempo, que explora sinergias entre o substrato de armazenamento e o paradigma publicação/subscrição, e permite consultas num escopo de tempo específico. De seguida, descreve em mais detalhe: i) Thyme, um sistema de armazenamento e disseminação de dados para ambientes sem fios na periferia da rede, que implementa ARCT; ii) Parsley, uma tabela de dispersão distribuída flexível e resiliente baseada em grupos, com realocação preventiva de nós e um mecanismo de particionamento dinâmico de dados; e iii) Thyme GardenBed, um sistema para armazenamento e disseminação de dados em redes multi-regionais na periferia da rede, que faz uso de interações entre dispositivos e com a periferia da rede.

As soluções desenvolvidas apresentam baixos custos, proporcionando tempos de resposta adequados para uso interativo e baixo consumo de energia, demonstrando serem práticas nas mais diversas situações. Estas soluções também exibem boas propriedades de balanceamento de carga e tolerância a faltas.

Palavras-chave: armazenamento de dados distribuído, disseminação de dados, computação na periferia da rede, publicação/subscrição, ponto-a-ponto, dispositivos móveis, redes sem fios

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AMC	adaptive multipart caching xxiii, 164, 165, 166, 175
AP	access point 14, 15, 28, 30, 31, 151, 152, 153, 154, 155, 156, 164, 192, 195
API	application programming interface 11, 52, 194, 195
AR	augmented reality 2
AWS	Amazon web services 167
BSS	basic service set 14
CBN	content-based networking 39
CCN	content-centric networking 39
CDN	content distribution network 31
CEP	complex event processing 29
CHR	cell hash routing 65, 72, 73
CRDT	conflict-free replicated data type 185
D2D	device-to-device xi, 10, 58, 100, 149, 150, 151, 152, 155, 156, 162, 168, 169, 170, 176, 180, 192, 195
DCS	data-centric storage 59, 65, 72, 98
DHT	distributed hash table xi, 9, 10, 11, 13, 30, 34, 35, 36, 37, 38, 41, 61, 71, 72, 103, 104, 105, 106, 107, 108, 109, 110, 115, 117, 118, 120, 123, 124, 127, 136, 145, 156, 180, 192, 193, 233
DNF	disjunctive normal form 69, 70
DNS	domain name system 36
DONA	data-oriented network architecture 39
DTN	delay tolerant network 30, 33, 41
ECA	event-condition-action 46, 47
ESS	extended service set 13, 14
GHT	geographic hash table xxi, 28, 59, 60, 61, 63, 65, 66, 67, 68, 69, 72, 74, 81, 97, 98, 99, 100, 192

gid	group identifier 110
GPSR	greedy perimeter stateless routing 72, 73, 77
IBSS	independent basic service set 14
ICN	information-centric networking 32, 38, 39, 41, 61
KBR	key-based routing 34, 104
LPC	local popularity cache 158, 160, 165, 166, 167, 173, 174, 175
MANET	mobile ad-hoc network 16, 29, 30, 40, 61, 181
MDD	mobile dynamic data set 183
MEC	mobile edge computing 4, 6, 152, 187, 188
NACK	negative acknowledgement 71, 74, 93
NDN	named data networking 39
NetInf	network of information 39
oid	object identifier 107, 119, 120, 121, 141
opkey	operation key 119, 120, 121, 122, 123, 141
ORC	other regions cache 159, 161, 162, 166
P/S	publish/subscribe xi, xxi, xxiii, xxv, 9, 10, 13, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 30, 32, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 53, 59, 60, 62, 63, 65, 66, 156, 159, 161, 171, 185, 189, 191
P2P	peer-to-peer 4, 8, 14, 23, 25, 31, 33, 54, 59, 65, 68, 100, 125, 181
PDS	peer data sharing 32
PPR	preemptive peer relocation 109, 115, 124, 125, 127, 134, 145, 146, 193, 233
PreC	prefetch cache 159, 160, 161, 162, 165, 166
PSIRP	publish-subscribe internet routing paradigm 39
RAN	radio access network 4
RPC	remote procedure call 35
RWP	random waypoint 92
sid	shard identifier 120, 121, 123, 124, 141
SSID	service set identifier 13
TARS	time-aware reactive storage xi, xxv, 9, 10, 11, 43, 44, 45, 46, 47, 48, 49, 50, 52, 53, 54, 57, 59, 62, 63, 98, 100, 155, 156, 184, 190, 191, 192

TTL	time-to-live 45, 110, 117, 162
TU	time unit 125
VANET	vehicular ad-hoc network 16
VR	virtual reality 2
WANET	wireless ad-hoc network 14, 15, 16, 18, 28, 32
WLAN	wireless local area network 15, 28
WMN	wireless mesh network 16
WSN	wireless sensor network 14, 16, 17, 28, 40
YCSB	Yahoo! cloud serving benchmark 136
ZRP	zone routing protocol 18

Chapter 1

Introduction

"Everything is bold to those who dare nothing."
— Fernando Pessoa

This Ph.D. thesis relates to the broad area of edge computing, focusing on its mobile aspect. More specifically, it explores resilient and efficient ways to provide data storage and dissemination among co-located mobile devices, both in settings with or without network infrastructure access. This first chapter characterizes the context and motivation for the conducted research work, as well as its main goals. Also, it presents the thesis' research statement and achieved contributions.

We start by giving some context and motivation in §1.1 and §1.2, respectively. Then, we present the main problem addressed in the thesis and its associated challenges in §1.3. Next, in §1.4, we define our research statement and its dichotomy, and describe the proposed approach and a summary of the achieved contributions. We conclude this chapter with §1.5, by presenting the outline for the rest of the thesis.

1.1 From Dumbphones to Smartphones

Since the advent of mobile computing [230], mobile applications have traditionally been devised under the de facto standard that mobile devices (e.g., smartphones, tablets) are *thin clients*. That is, very resource-constrained devices that serve primarily for data gathering (e.g., user or sensor input), relying on back-end servers to do all the work and respond back with the final result. This was an intrinsic idea to *mobile cloud computing* [91] for some time, whereby data storage and processing happened mainly *outside* the mobile devices, while only a lightweight front-end application ran on them—a technique usually referred to as computation offloading [54, 63].

However, we are witnessing a rapid and steady growth of both the capabilities and amount of mobile devices worldwide [56]. With octa-core, 8+ RAM mobile devices readily available on the market [119, 189], today's devices have more computational power than the desktop computers of a few years ago [52]. Likewise, the extensive proliferation of these devices is making them increasingly ubiquitous, with their number expected to exceed 13 billion by 2023 [59]. Thus, we can start harnessing those resources and make a more judicious *offload* of only certain heavy computations.

As a consequence of these advancements, we are seeing a paradigm shift to a more mobile world. In the past years, there has been a big rise in the adoption of mobile devices for communicating with friends, performing daily activities, or even perform work-related tasks. Both Cisco and Ericsson forecast that mobile data traffic will grow to roughly 77 *exabytes* per month worldwide by 2022, a 11× increase comparing to the seven monthly exabytes in 2016 [56, 58, 81]. Furthermore, analysts convey that the exponential growth of mobile data traffic is directly related with the increase in user-generated content, such as video (which will account for 80% of all mobile traffic by 2022 [58]), as well as the arising of next generation resource-hungry applications [109], like augmented reality (AR) and virtual reality (VR).

Despite all the developments in the mobile computing area, those next generation applications are resource-intensive and latency-sensitive, even for what today's devices and networks can deliver. Furthermore, concerning mobile devices, even if they can deliver the required computing power (and other resources), the trade-off is *extremely short* battery life [115, 173, 275]. Additionally, the wireless communication technologies used by these devices are unreliable and congestion-prone (e.g., Wi-Fi, 4G) [3, 125].

Likewise, caused by the sheer amount of data coming from those applications and user-generated content [201], the strain in the core networks is becoming an issue. Usually, the back-end services that support these use cases leverage on cloud data centers to process and store data. Even though these infrastructures offer various benefits (e.g., elasticity, massive redundancy, geo-replication), their *consolidation* and *centralization* represent a *large separation* between mobile devices and data centers, resulting in high latencies and low bandwidth [78, 110, 117]. Altogether, interactions between mobile devices and distant cloud infrastructures can be *unfeasible* in certain situations and *unwanted* in others, thus establishing the need for distributed cloud services *closer* to the end-user devices—at the *network edge* (as opposed to the network core, where data centers reside).

1.2 One Step Closer to the Edge

Cloud infrastructures became the prominent focal point of every network, connecting applications, services, and devices together. However, the billions of connected devices worldwide, generating and exchanging enormous amounts of data, impose new challenges that make it really difficult to comply with the requirements of certain (mobile) applications and start to surpass the capability of cloud computing infrastructures.

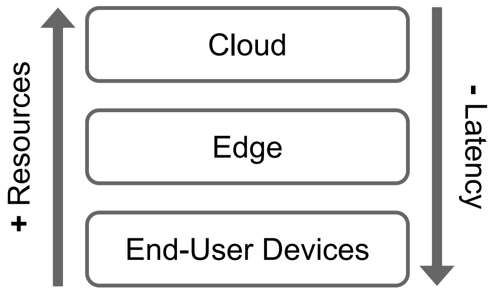


Figure 1.1: Simplified network hierarchy in the edge-centric paradigm.

As such, several approaches (that we present ahead) proposed the idea of relocating some of the cloud's capabilities closer to the end-user devices. Cloud services are brought to the physical edge of the network, providing an intermediate layer in the network hierarchy—end-user devices, *edge*, cloud—and shortening the distance to such services, as in Figure 1.1. The *edge-centric* paradigm [96, 240] was proposed to *complement* the cloud computing model, exploiting resources available at the network edge and avoiding the need to execute applications fully on the cloud (thus, possibly entailing reduced latencies). Nonetheless, if necessary, complementary processing and storage can still be performed at cloud data centers (e.g., for archival or analytics purposes), where resources are more plentiful and powerful.

This paradigm can be seen as a distributed extension of the cloud computing model, whereby the cloud core infrastructure is broken down into a network of smaller "clouds" located close to the end-users. Since data is stored and processed close to its source, this approach is able to be more responsive while reducing some of the load from both cloud and network infrastructures, also yielding other benefits such as privacy or availability.

Although similar concepts have been proposed, such as cloudlets [231, 233, 262, 280], fog computing [37, 121], and edge computing [96, 240], they all revolve around the same idea of leveraging edge resources to perform computations and store data close to its source, but each having its idiosyncrasies[1] [232, 276, 287].

A *cloudlet* can be seen as a "data center in a box" located at the edge of a network, thus bringing the cloud closer to the end-users. It is usually based on dedicated small-scale servers, ideally located one wireless hop away from the devices, e.g., in a cellular or Wi-Fi base station. Then, applications running on the devices connected to these stations can harness these low-latency computing and storage resources.

As a cloud closer to the ground, *fog computing* brings processing capabilities down to the local area network. Fog nodes (e.g., gateways, routers, access points, switches) gather, process, and store data from multiple sources within the network. After processing, pertinent data (and any additional information) is transmitted back to the necessary devices (or to the cloud for further processing or long-term storage).

[1]Sometimes, even the same concept can have multiple slightly different definitions proposed by distinct authors.

In turn, *edge computing* brings processing capabilities even further down, directly into the end-user devices[2]. Instead of sending data to centralized entities in the local network for processing (e.g., fog nodes), devices collect, analyze, and process data they generate. Each device is independent of the rest and is capable of deciding what data should be processed and stored locally, and what needs to be sent up the hierarchy (e.g., to the cloud) for further use.

When compared to edge computing, fog solutions have the advantage of being able to see the "bigger picture", since they process data received from multiple sources. However, edge computing leverages on the processing power of the end-user devices themselves, thus it has inherently less (central) points of failure. Since it does not require the use of dedicated network nodes, edge computing also reduces system and network architecture complexity. Cloudlets are similar to fog nodes, but usually this term is used to refer to dedicated servers (e.g., mini data centers) connected to base stations.

With the arrival of 5G networks [18], network providers and telecommunication companies also brought the edge-centric paradigm into their domain. Mobile edge computing (MEC)[3] applies the ideas of cloudlets, fog and edge computing into the radio access network (RAN). It provides computing and storage capabilities at the edge of the cellular network, in close proximity to the mobile subscribers [118]. It also allows network operators to open their RAN to authorized third-parties, such as application developers and content providers, by offering application and service hosting [154].

Another related concept is *mobile edge cloud* [78]. In many situations, mobile users gather in a localized geographical area for some time (e.g., sporting events, conferences) and use their mobile devices to access cloud services through Wi-Fi or cellular networks. However, as already referred, both the amount and capabilities of mobile devices have been increasing largely in the past years, and that growth does not seem to slow down. Thus, it starts to become a possibility to obtain some of these traditional cloud services locally from nearby mobile devices, using self-organized *ad-hoc* networks formed among such devices. These clouds are of particular interest in low connectivity scenarios. A single device may not own enough resources to carry out certain tasks, but collectively co-located devices can provide sufficient capacity to satisfy the transient computational and storage needs of the local users. Hence, these clouds harness the collective resources of a group of mobile devices in close geographical proximity, collaborating together to form opportunistic and ephemeral clouds that are very cost-effective but highly volatile.

In the end, the edge-centric paradigm, in all its forms, posits taking the control of applications, data, and services away from the core, to the other end of the network. It extends the concept of peer to all the devices at the network edge, presenting itself as the natural confluence of peer-to-peer (P2P) and cloud computing to create hybrid architectures that combine stable resources with mobile terminals [96].

[2]At least, in this document, we use this notion of edge computing, as depicted in Figure 1.1.

[3]More recently, the meaning of MEC was changed to now mean *multi-access* edge computing, in order to reflect that the edge is not only based on mobile networks.

1.3 The Zettabyte Era

The pervasiveness of mobile devices makes them the primary tool for generating and sharing all sorts of content (e.g., video, photos) [56, 59]. Accompanying that trend, there is a big demand for the in loco (real-time) sharing and dissemination of content people generate in all kinds of social gatherings (e.g., concerts, sports events) [57, 82]. Thus, mobile users expect to use their devices (almost continuously) to both access and share those contents. However, this usage pattern places a huge burden on network infrastructures and cloud-based services alike, because they have to accommodate high loads to support that continuous user activity [82].

The typical alternative to sustain such high demand is to set up special network infrastructures just for those events. Unfortunately, in some scenarios it might be logistically or financially nonviable to deploy such setups. Communication infrastructures in crowded venues are known to sometimes be overloaded or provide low or intermittent connectivity [82]. In one-time events (e.g., conferences, reunions), it may not pay off to set up communication infrastructures just for those occasions. In disaster situations, communication infrastructures may not even exist (e.g., they could have been destroyed) or may not be feasible to set up [166].

Since the major part of this content is often (centrally) stored and processed at cloud infrastructures, many concerns arise about how these extreme volumes of data should be transferred, stored, processed, and made available. Such centralized solutions may lead to privacy concerns, violate the latency constraints of mobile applications, or may be infeasible due to bandwidth or energy constraints of mobile devices. Thus, resorting (only) to cloud infrastructures to support such uses cases can become unfeasible due to the unprecedented amount of generated data. The transfer of such large amounts of data to the cloud can lead to network congestion, processing delays, and possible monetary costs. Additionally, in several scenarios (e.g., crowded venues [82], disaster situations [166]), mobile devices often experience poor or intermittent connectivity, leading to availability issues if application storage and logic are fully delegated to a remote cloud infrastructure.

Considering that analysts predict mobile data traffic will have an exponential growth in the coming years [56, 57, 59, 81], those huge amounts of generated data are becoming an issue, both from the network infrastructures and cloud services perspectives. Thus, it is best if we start harnessing the available resources at the network edge, and disperse (and process) those extreme amounts of data among the different levels of the network hierarchy.

With the appearance of the edge-centric paradigm [96], data generated at the edge of the network can be processed and stored near its source. Thus, diverting some of this data from the cloud, and relieving some of the load from both cloud and network infrastructures. Avoiding data uploads to a centralized entity can not only help to preserve privacy, but also to reduce network traffic congestion. However, this distributed and collaborative data storage and processing still requires communication between devices (and possibly other entities) over wireless links. Thus, the unreliability of wireless channels can significantly affect the operation of edge solutions. Additionally, edge resources are not as powerful as the ones in cloud infrastructures, i.e., we will not have high-end server racks attached to base stations. Thus, the usage of such resources should be made judiciously, so as not to overload them as well.

The general problem we address in the thesis is how to allow resilient and efficient data storage, dissemination, and querying among co-located mobile devices, both with or without access to network infrastructures and MEC capabilities. From a high-level perspective, several challenges arise, such as what data to store persistently and what data to cache; where to place what data; how and when to propagate data (and metadata); how to query data and how to make it available to stakeholders; etc. Furthermore, several other broad challenges typical of these dynamic environments ensue, including:

- support decentralized and loosely coupled settings (thus, with a lack of global state);
- allow a large number of heterogeneous (mobile) devices;
- efficiently detect and tolerate high membership dynamics (i.e., devices' mobility and churn);
- support intrinsically asynchronous environments;
- ensure data persistence;
- provide low latencies;
- minimize energy consumption; and
- endure poor/intermittent connectivity (i.e., frequent network partitions).

In the end, we design all our proposals in order to allow data storage and dissemination in pervasive edge computing environments with the mentioned characteristics. At the same time, aiming at tackling all these challenges in a holistic way, as we will describe in the next sections and chapters.

1.4 Data Storage and Dissemination at the Network Edge

The rise of the edge-centric paradigm, specifically directed to mobile ecosystems, leads to a key insight. *It is more efficient to communicate and distribute information among nearby devices than to use distant centralized intermediaries* [78, 110, 117]. This remark is even more exacerbated when referring to mobile devices, known for having some constraints, mostly regarding energy and communications. Note that when referring to mobile ecosystems, mobile edge environments, or pervasive edge computing environments we mean networks comprised of (possibly co-located) wirelessly connected mobile devices (using either Wi-Fi, 3G/4G, etc., i.e., multi-access environments).

Our main goal is to provide resilient and efficient data storage and dissemination in pervasive edge computing environments, either with or without access to network infrastructures. On the one hand, it should be able to operate in environments where there is no access to network infrastructures, thus working in ad-hoc settings (although that maybe offering only a subset of its features). On the other hand, it should operate in environments with access to network infrastructures, leveraging judiciously on the available resources.

As an example, we envision our solutions possibly being used in a plethora of scenarios, such as: i) remote locations; ii) natural parks; iii) one-time events; iv) amusement parks; v) protests; vi) university campuses; vii) sports fan zones; viii) disaster scenarios (e.g., natural disasters, or search & rescue operations); ix) hacker attacks (e.g., denial of service); x) crowded venues; or xi) censorship attempts (e.g., Hong Kong riots). In these cases, some scenarios do not have network infrastructures at all (e.g., i and ii), others might have them only in some parts of the venue (e.g., iii and iv), and some usually have an ample network infrastructure setup (e.g., vi and vii). In some situations, although network infrastructures may exist, these may not be usable (e.g., viii, ix, and x), or we may not want to use them (e.g., xi).

1.4.1 Research Statement

In line with our goal, the thesis explores the following research question: **How to support resilient and efficient data storage and dissemination solutions in pervasive edge computing environments, operating with or without access to network infrastructure?**

In our solution, we argue for a distributed hyper-local data storage and dissemination system for this kind of environments. The hyper-local adjective connotes information oriented around a community, with its primary focus directed towards the concerns of its members. A key insight behind this hyper-local data storage solution is that users who convene for all kinds of social gatherings are usually interested in similar types of information [254] (e.g., statistics and videos at sports events, coupons at shopping malls). Following this, data can be kept locally to reduce transmission bandwidth and latency (also reducing the load in cloud core infrastructures).

This work can be further divided into two more specific questions, exploring both aspects of our goal. The first one relates with the absence of network infrastructure and the provision of transient data storage and dissemination for networks of co-located mobile devices: **How to support reliable and efficient data storage and dissemination in wireless edge environments without access to any kind of network infrastructure?** In certain scenarios, network infrastructures may not exist or may be inoperable. For instance, information dissemination and sharing can be of great use in remote locations without network coverage or in disaster scenarios where network infrastructures were destroyed [166]. In these settings, the storage service can be classified as transient or ephemeral, in the sense that it will only exist while there are devices supporting it.

The second specific question relates with the existence of access to network infrastructures and their corresponding edge computing capabilities: **How to leverage on edge computing capabilities to improve the performance, scalability, and resource management of the previous solution?** With the existence of infrastructure access, and the rise of the edge-centric paradigm, the data storage and dissemination system can be extended to take advantage of such capabilities. It can leverage on this new level of the network hierarchy and store data in its different levels, each with its own specific resources and guarantees. Additionally, it can also share some (management) workload between the hierarchy levels.

Following from the defined research questions, our main research statement is:

It is possible to provide resilient and efficient data storage and dissemination solutions for pervasive edge computing environments, able to operate with or without access to network infrastructure.

Since network infrastructures are not always available or accessible, and according to the research statement, we adopt the following motto for the thesis:

"Surviving without infrastructure. Thriving with infrastructure."

When they exist, in most cases we may leverage on the available resources, but we should be able to survive without them, in infrastructure-less settings.

1.4.2 Proposed Approach

Figure 1.2 depicts an overview of our proposed architecture, where it includes scenarios with (scenarios A, B and D) and without (scenario C) network infrastructure access. For the scenarios without network infrastructure access, the storage solution works in a purely wireless ad-hoc environment, where the devices themselves contribute with their computing and storage resources to the system. Devices work in a P2P fashion, and communication among them is achieved using multi-hop routing. In turn, for the scenarios with access to network infrastructure, we take advantage of the existing edge computing capabilities. As such, we have at our disposal a three-tier hierarchy, each providing different guarantees and resources. Nonetheless, in this work, we only address the two lower levels of the network hierarchy depicted in Figure 1.1 (i.e., the end-user devices and the edge). We have current work that is starting to explore the third and topmost level of the hierarchy (i.e., the cloud), trying to vertically integrate all the levels and reap the advantages of each one.

We target highly dynamic and asynchronous environments, where devices can move freely, and network links often go down intermittently and have limited bandwidth (e.g., due to wireless channel errors). Additionally, mobile devices are battery-powered, thus placing further constraints on communication and processing capabilities. Such resource-constrained and highly dynamic environments are challenging for tightly coupled distributed applications, i.e., in these conditions, the use of traditional client-server solutions

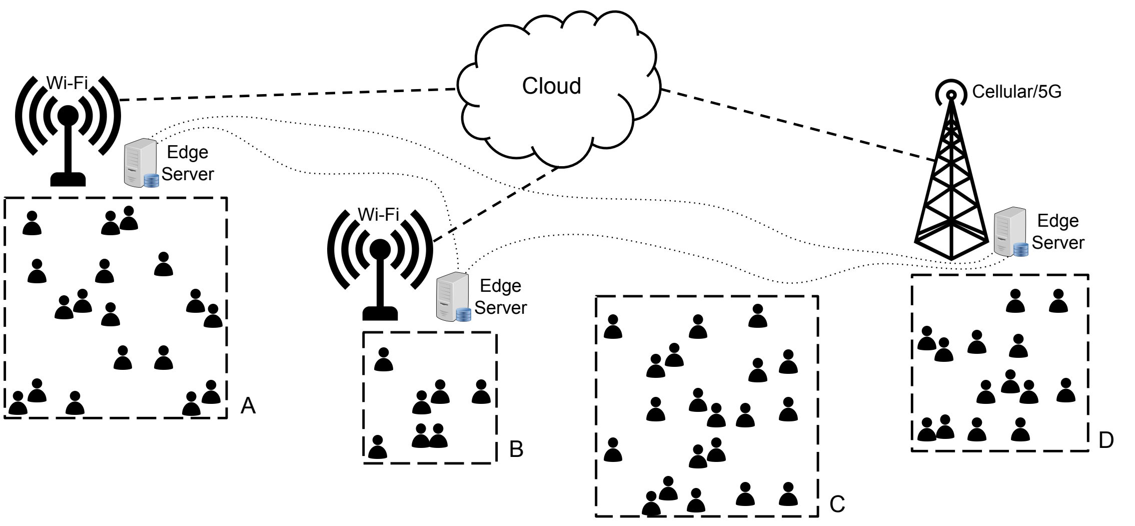


Figure 1.2: Overview of the thesis proposed architecture.

that rely on server accessibility should be avoided. In turn, symmetrical (distributed) architectures making use of asynchronous communication are more robust and can better tolerate these transient disconnections.

Accordingly, we favor a corresponding loosely coupled approach for achieving data dissemination. Hence, we fuse the storage substrate with the publish/subscribe (P/S) paradigm and propose *time-aware reactive storage (TARS)*, a reactive data storage and dissemination model. Following this model, users register their interests through *subscriptions* within a *specific time scope*. Subsequently, they are notified as new relevant data is stored in the system through *persistent publications*.

1.4.2.1 Surviving Without Infrastructure

Either because they were destroyed, there are too many users trying to connect, or simply because they do not exist, network infrastructures may not be accessible in every situation. Still, in many scenarios (e.g., disaster situations, crowded venues, remote locations), making information available can be of paramount importance [70, 166]. Thus, this part of the thesis relates with the development of a transient data storage and dissemination system for networks of co-located mobile devices in the absence of network infrastructure.

Here, we propose Thyme, a data storage and dissemination system for wireless edge environments, that implements TARS. Essentially, it is a system that makes opportunistic use of mobile devices and ad-hoc networking to provide a transient storage service in a localized geographical region. We pursue two different paths. One follows a lightweight *unstructured* approach using local storage and query flooding, named Thyme-LS. The other, named Thyme-DCS, embraces a more intricate *structured* approach, specifically a geographical distributed hash table (DHT), lead by the fact that geographic positions have a close relation with the topology in wireless networks. We implement both approaches in a network simulator [216], and the DHT approach also as an Android library.

1.4.2.2 Thriving With Infrastructure

With access to the network infrastructure and the rise of the edge-centric paradigm, our solution can be extended to take advantage of the guarantees and resources available at various levels of the network hierarchy. Thus, this part of the thesis relates with the development of data storage and dissemination solutions for networks of co-located mobile devices with access to network infrastructures and their corresponding edge computing capabilities. We harness the resources available at the network edge in order to provide scalable and flexible data storage and dissemination, while still keeping data close to the end-users. By exploiting different levels of the network hierarchy, we can push some of the work out of the mobile devices and into the upper levels of the hierarchy, sharing some management responsibilities.

Here, we propose two different solutions. The first one addresses some challenges in managing highly dynamic device population and workload imbalances in the context of structured overlays, namely DHTs. To tackle these issues, we propose Parsley, a resilient *group-based* DHT with preemptive *peer relocation* and a dynamic *data sharding* mechanism. We implement this proposal in the PeerSim simulator [180].

The second one, Thyme GardenBed (implemented and evaluated in the context of the M.Sc. of Vieira [278]), concerns a data storage and dissemination system for *multi-region* edge networks (where we leverage on our previous work, Thyme). We take advantage of edge servers to *cache* some (popular) data and perform some of the system's management, interacting among each other. Devices cooperate among each other and with the edge servers, and *share* storage and management responsibilities. We implement this approach as an Android library (and experiment with real devices and simulation).

1.4.3 Contributions

The thesis' main contributions are distributed data storage and dissemination solutions for pervasive edge computing environments, able to operate in settings with or without network infrastructure access. Namely, it presents the following contributions:

1. Time-aware reactive storage (TARS), a reactive data storage and dissemination model with intrinsic time-awareness, that fuses a P/S abstraction with the storage substrate, and allows queries within a specific time scope;

2. Thyme, a data storage and dissemination system for wireless edge environments, implementing TARS;

3. Parsley, a flexible and resilient group-based DHT with preemptive peer relocation and a dynamic data sharding mechanism; and

4. Thyme GardenBed, a framework for data storage and dissemination across multi-region edge networks, that makes use of both device-to-device (D2D) and edge interactions (implemented in the context of the M.Sc. of Vieira [278]).

1.5 Document Outline

The remainder of the thesis is organized as follows:

- **Chapter** 2 introduces fundamental concepts and relevant state of the art required for the better understanding of the contributions presented in the thesis;
- **Chapter** 3 presents the time-aware reactive storage (TARS) concept in detail and its application programming interface (API);
- **Chapter** 4 details Thyme, a system implementing TARS for wireless edge environments, its two different approaches (Thyme-LS and Thyme-DCS), and the Android implementation of Thyme-DCS;
- **Chapter** 5 describes Parsley, a group-based DHT with preemptive peer relocation and a dynamic data sharding mechanism;
- **Chapter** 6 explains Thyme GardenBed, a framework for content storage and dissemination across multi-region edge networks;
- **Chapter** 7 presents a brief overview of works related with the thesis, some of which are based on and derived from Thyme, exploring different research directions (in what we called the Edge Garden ecosystem); and
- **Chapter** 8 concludes the thesis by summarizing the achieved results, and discussing several pointers for future research directions.

Additionally, at the end of the thesis, Appendix A describes a characterization study of the group size parameters of Parsley.

Chapter 2

Research Context

"If I have seen further than others, it is by standing upon the shoulders of giants."
— Isaac Newton

The thesis addresses data storage and dissemination in pervasive edge computing environments. In order to better understand its content, it is essential to be aware of its relevant subjects. To this extent, in this chapter, we present the research context and some related work which contextualize our work.

In §2.1, we characterize the wireless environments targeted by the thesis. Next, in §2.2, we survey concepts around the publish/subscribe (P/S) paradigm and present some related work. Then, in §2.3, we review previous research regarding data storage and dissemination in pervasive edge computing environments. In §2.4, we present some concepts regarding overlay networks, and specifically distributed hash tables (DHTs). After, in §2.5, we give a brief presentation about some other topics relevant to the thesis (namely, other data dissemination models). Lastly, we wrap up with §2.6, where we present our final considerations regarding the surveyed matters.

2.1 Off the Wire: A Primer on Wireless Networks

A wireless network uses a wireless transmission medium for the exchange of information, enabling two or more devices to communicate among them without using network cables, i.e., without a physical connection [97, 199]. These networks are usually implemented using radio-based transmission, the dominant form of wireless transmission.

In wireless networking standards (e.g., IEEE 802.11), an *extended service set (ESS)*, or just service set, is a group of wireless devices which are identified by the same service

Table 2.1: Differences between infrastructure and infrastructure-less networks [227].

	Infrastructure network	Infrastructure-less network
Structure	Fixed	Non-existent
Topology	Static backbone	Highly dynamic
Connectivity	Stable	Irregular
Setup cost	High	Low
Setup time	Large	Small

set identifier (SSID), forming a logical network (i.e., the devices are on the same logical network segment) [97]. In turn, *basic service sets (BSSs)* are sub-groups of devices within a service set which are operating with the same physical layer medium access parameters (e.g., radio frequency, modulation scheme, security settings) such that they are wirelessly networked. Thus, the BSSs of an ESS appear as a single network to the logical link control layer, meaning that devices within an ESS can communicate with each other, and can even move freely and transparently between BSSs (of the same ESS, naturally). There are two categories of BSSs. The ones formed by an *infrastructure* mode redistribution point—e.g., an access point (AP)—, and those that are formed by independent stations in a peer-to-peer (P2P) *ad-hoc* topology—an independent basic service set (IBSS). Note that a BSS should not to be confused with the coverage area of an AP, known as the basic service area.

An *infrastructure* network resorts to an AP where client devices wirelessly connect to and communicate through. That is, clients communicate only with the AP they are connected to, and all traffic within the BSS is routed by that redistribution point. APs define the BSS operating parameters, and are usually connected to a wired backbone, working as gateways to other networks. Examples of this kind of networks include cellular and Wi-Fi local networks. In turn, an *infrastructure-less* network, also called wireless ad-hoc network (WANET), has no infrastructure (redistribution point) and is entirely wireless, i.e., (client) devices communicate directly with each other in a point-to-point fashion. If a device wants to communicate with another one outside of its radio coverage, that message will have to be routed by other devices in the network in a multi-hop way. A wireless sensor network (WSN) is an example of such network.

Table 2.1 presents the main differences between these two types of networks. In an infrastructure network there is a base station that administers the wireless devices connected to it, giving it a fixed structure and stable connectivity. On the contrary, in a WANET, devices dynamically form a network without the use of any existent infrastructure or centralized administration, resulting in a more cumbersome structure and irregular connectivity [227]. While setting up WANETs is very cost-effective and quick, for infrastructure networks it entails high costs and is time-consuming.

Wireless networks are usually affected by some issues that arise from the characteristics of their communication medium. They are frequently subject to interferences caused by radio waves generated by other networks, degrading the signal or even causing communications to fail. They are also affected by some materials that absorb or reflect radio waves, preventing them from reaching the intended receivers. Also, the hidden node problem occurs when a node A is visible from another node B, but not from other nodes communicating with node B, thus leading to difficulties in controlling the access to the wireless medium. Since the wireless spectrum is a limited resource, bandwidth needs to be shared among the multiple users, resulting in lower individual user rates.

2.1.1 Wireless Infrastructure Networks

For providing the communication channel, these networks use fixed wireless hubs—base stations or APs—where devices connect to and communicate through. Two of the most known infrastructure networks are cellular and wireless local area networks (WLANs).

As its name suggests, a *cellular (or mobile) network* is a communication network distributed over land areas called cells, each served by at least one fixed base station (e.g., a cell tower) [174]. These base stations provide the cell with the network coverage which can be used for data transmission. A cell typically uses a different set of frequencies from neighboring cells, to avoid interferences and provide guaranteed quality of service within each cell. Since cell towers are close to mobile devices, the devices use less power than with a single transmitter or satellite. When joined together, cells provide radio coverage over a wide geographic area (in the order of several kilometers), thus enabling a large number of devices (e.g., mobile phones) to communicate with each other, even if some are moving through different cells during transmission.

A *WLAN* is a wireless network used to connect devices in a limited area, such as a home, school, or office building [97]. Most modern WLANs are based on IEEE 802.11 standards and are marketed under the Wi-Fi brand name. Through the AP, it can also provide a connection to other networks (including the Internet and non-wireless devices).

2.1.2 Wireless Ad-Hoc Networks

A WANET is a wireless network that does not rely on a preexisting infrastructure. Instead, the nodes forming the network forward messages on behalf of others [51, 227], where these forwarding decisions are made dynamically, following the routing protocol deployed in the network. Since these networks are decentralized, they are dynamic self-configuring networks in which the majority of the nodes are free to move.

Such decentralized nature makes them suitable for scenarios where a central coordination point cannot be relied on, either because it is impossible to deploy, or it cannot be trusted. They can be applied in different areas such as disaster relief, environmental monitoring, or military communications. For instance, in military scenarios, due to its infrastructure-less nature and fast deployment, these networks are used by military units (e.g., soldiers, drones, ships) to communicate in harsh terrains, coordinate in battlefield operations, and share crucial information (e.g., imaging, multimedia data) [70].

Also, during natural disasters, a quickly deployable communication channel is of extreme necessity, even more when traditional communication infrastructures are destroyed [166]. Rescue teams can use these networks to communicate and exchange vital information. It can also be used to extend the range of wireless infrastructure networks, where some nodes can work as gateways into the infrastructure network.

The flexibility of these networks comes at a price. They are highly dynamic, network topology can change very frequently (because of user mobility or failures), network links often have intermittent connectivity, and their bandwidth is limited because of wireless channel errors (usually an order of magnitude lower than wired links; for long multi-hop connections bandwidth reduction is even more dramatic) [227]. Here, communication failures are the norm rather than the exception.

A *mobile ad-hoc network (MANET)* consists of mobile devices that communicate among each other in a wireless fashion, for instance, using the IEEE 802.11 standard [28]. Those devices spontaneously and autonomously form a network amongst themselves without any central infrastructure or fixed topology. Devices are free to move independently from each other, thus topology is highly dynamic.

Wireless mesh networks (WMNs) are a particular type of WANET where nodes are of two types: mesh routers and mesh clients [8]. Mesh routers are dedicated wireless devices (usually without energy restrictions) and have minimal to zero mobility, thus topology tends to be static. They form a multi-hop wireless backbone, and some might act as gateways to the Internet (or other networks). In turn, mesh clients connect to the routers. Consequently, clients mobility or energy issues do not affect the backbone network topology. Mesh clients are often laptops, cellphones, and other wireless devices. An example of such network is Guifi.net [104].

Vehicular ad-hoc networks (VANETs) apply the same principles of MANETs to vehicles moving on roads. They are used for communication between vehicles and roadside equipment [260]. Typically, power consumption is not an issue in this type of network. Rather than moving freely at random, vehicles tend to move in a more organized fashion, since they are constrained to follow roads. Since the domain of VANETs is very restricted, usually its applications are targeted towards traffic information systems and intelligent transportation systems. For instance, it can be used to provide real-time obstacle reports, road safety warnings, and traffic status information.

Sensor nodes are small, low-power inexpensive devices, with limited computing resources [288]. They can sense, measure, and gather information from the environment they are in, e.g., sound, temperature, humidity, pressure. A *wireless sensor network (WSN)* typically has little or no infrastructure. It consists of a number of spatially dispersed sensor nodes, working together to monitor a region and obtain data about the environment. Since they are typically deployed in difficult-to-access locations [129], wireless communication is used to transfer the sensed data outside of the sensor network, to a sink node or gateway for further processing and analysis. Sensor nodes can communicate directly among themselves, and typically pursue multi-hop paths to disseminate the collected data towards the network sink node(s). Some applications of WSNs are area monitoring, fire detection, animal tracking [129], or pollution monitoring.

Also called disruption tolerant networks, *delay tolerant* or *opportunistic networks* are an approach that seeks to address the issues in heterogeneous networks characterized by very long delay paths and frequent network partitions [89]. These networks look at mobility, disconnections, partitions, etc. as features of the networks rather than exceptions. In fact, mobility is exploited as a way to bridge disconnected cluster of nodes. Examples of such networks are those operating in extreme terrestrial environments or interplanetary networks. Since instantaneous end-to-end paths many never exist, routing protocols have to take a store-and-forward approach [124], where data is incrementally moved and stored throughout the network in hopes that it will eventually reach its destination.

2.1.3 Routing Protocols

A routing protocol is a specification of how routers decide to forward packets among each other. In wireless ad-hoc networks, every node is a potential router and topology is dynamic. Thus, routing protocols need to define what information needs to be exchanged to enable them to build (and keep up-to-date) some knowledge base of the network topology. Some inherent constraints in wireless environments are low bandwidth, limited energy, high error rates, asymmetric links, and other factors such as terrain conditions, obstacles, etc.

There are two approaches for ad-hoc routing protocols: topology- and position-based routing. Topology-based protocols can be further divided into proactive, reactive, and hybrid strategies.

2.1.3.1 Proactive Protocols

Also called table-driven, *proactive protocols* employ classical routing strategies such as distance-vector (e.g., DSDV [200]) or link-state routing (e.g., OLSR [123]). They continuously maintain up-to-date routing information about all the available routes in the network (through periodic dissemination), even if these are not currently used. Routing tables are periodically disseminated throughout the network in order to maintain them up-to-date. The main drawback of this approach is that the maintenance of unused routes may occupy a significant part of the available bandwidth and drain nodes' battery even when the network is idle, or if the network topology changes frequently.

2.1.3.2 Reactive Protocols

Reactive protocols, also known as on-demand protocols, only maintain the routes that are currently in use, thus avoiding the burden of maintaining unused routes. Since only used routes are maintained, prior to packet forwarding, nodes are required to perform a route discovery phase, thus leading to a route setup delay before the actual packet routing. Even though only used routes are maintained, this can still represent a significant amount of network traffic when network topology changes frequently. Examples of reactive protocols are AODV [198] and DSR [128].

2.1.3.3 Hybrid Protocols

As its name suggests, *hybrid protocols* combine the main ideas of reactive and proactive approaches, trying to bring together the advantages of both. Typically, they try to exploit the low communication overhead of reactive protocols and the reduced route setup delay of proactive protocols. An example is zone routing protocol (ZRP) [111]. Briefly, it defines a zone around each node, that contains the neighbors within a given number of hops. Then proactive algorithms are used to route packets within the zone, whereas reactive algorithms are used to route packets outside the zone. However, even a combination of both strategies requires to maintain the currently used routes, limiting the amount of topological changes that can be tolerated within a given period of time.

2.1.3.4 Geographic Protocols

Also named position- or location-based, *geographic protocols* use nodes' location information to find the best routes between source and destination [169]. The location information can be obtained from various sources, such as GPS or other positioning service [50, 206]. Without knowledge of the network topology or a prior route discovery, at each hop, the routing decision is based on the destination's position and the position of the forwarding node's neighbors. The position of a node's neighbors is typically learned through periodic beacons (containing the sending node's position).

These protocols do not need to establish or maintain routes, thus nodes do not have to store routing tables nor have to transmit messages to keep them up-to-date. Instead, they congregate some ideas of both reactive—calculate routes on-demand—and proactive routing protocols—very small periodic beacons. The simplest routing strategy is *greedy forwarding*, whereby a message is forwarded to the node that minimizes (at each hop) the distance to the message destination. Other strategies have to be used when greedy forwarding is not possible, such as *perimeter forwarding* [135].

It has been verified that topology-based routing protocols are not scalable [257]. In large networks, geographic protocols deliver more packets and consume less network resources than topology-based approaches. Topology-based routing requires the maintenance of (somewhat) accurate topology information of the network. The communication overheads for such information maintenance quickly increase with the network size and the amount of topology changes. In turn, geographic protocols avoid those overheads by requiring only localized information, therefore being aligned to provide better overall performance for large networks. Also, regarding resource-constrained (mobile) devices, position-based routing will probably deliver the best trade-off between expended resources and routing performance, when in WANET scenarios.

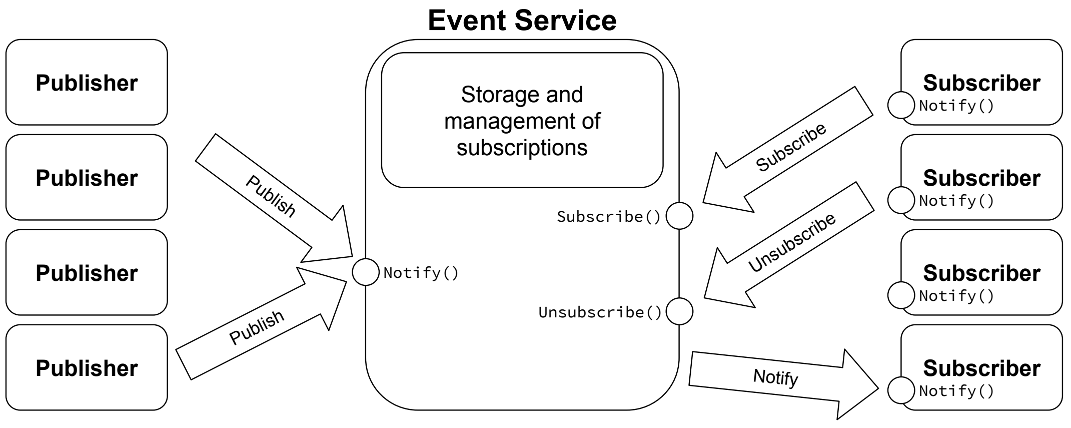


Figure 2.1: A basic publish/subscribe system model [85].

2.2 The Hitchhiker's Guide to Publish/Subscribe

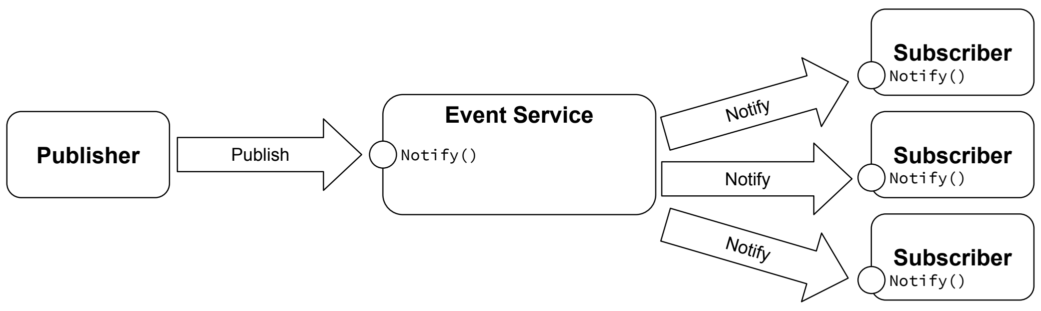
The publish/subscribe (P/S) interaction paradigm provides a simple, yet effective, communication abstraction, allowing the asynchronous exchange of information from producers to consumers. In this event-based paradigm, data sinks, usually called *subscribers*, express their interest in an event or class of events. Subsequently, they are notified of any event, generated by a data source (usually referred to as *publisher*), that matches their registered interests [85, 187]. In other words, publishers publish information, categorizing it into classes, and without knowing which subscribers there may be, if any. In turn, subscribers subscribe to the information categories they are interested in, and get notified only about those. This information is typically denoted by the term *event* and the act of delivering it by *notification*. A subscriber registers its interest in events through *subscriptions* and the act of generating an event is denoted as *publication*.

The basic P/S system model, illustrated in Figure 2.1, relies on an *event notification service* that manages subscriptions and delivers events, acting as a mediator (or a proxy) between publishers and subscribers. A subscriber calls a `subscribe()` operation to register its interest in certain events. The symmetric `unsubscribe()` operation revokes a subscription. In turn, a publisher calls a `publish()` operation to generate an event. Then, the event service propagates the event to all relevant subscribers.

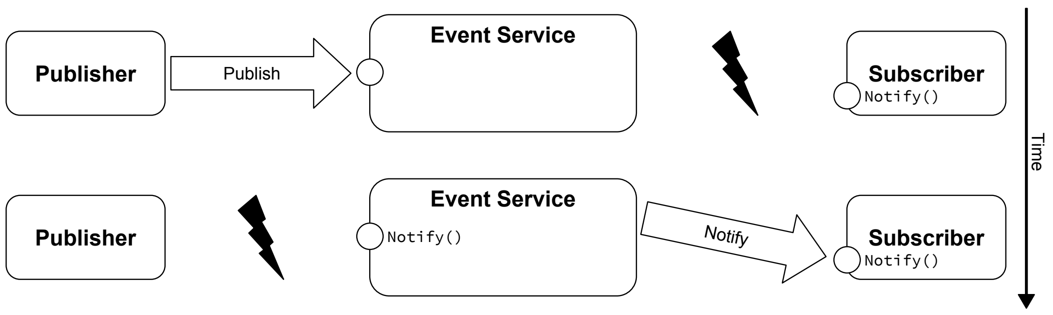
2.2.1 Loose Coupling

As the event service acts as a proxy between publishers and subscribers, it provides full decoupling among all the communicating entities in *space*, *time*, and *synchronization* [85].

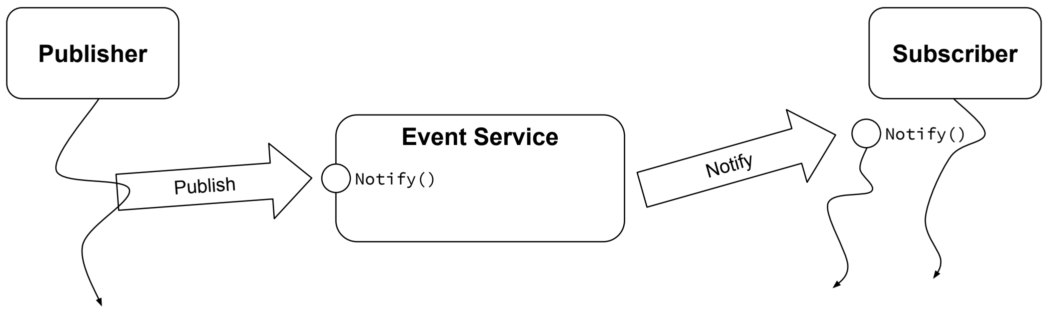
By removing all explicit dependencies between the interacting parties, the P/S paradigm is well adapted to the loosely coupled interactions required in many (large-scale or highly dynamic) distributed environments—which are asynchronous and volatile by nature. The P/S loosely coupled interaction model is depicted in Figure 2.2, through its three decoupling dimensions.



(a) Space decoupling.



(b) Time decoupling.



(c) Synchronization decoupling.

Figure 2.2: Publish/subscribe decoupling dimensions [85].

Space Decoupling. Publishers and subscribers interact without knowing the existence of each other, i.e., the event notification service presents itself as an anonymous communication channel. A publisher publishes events through the event service, and subscribers are notified about those events indirectly through the same event service (Figure 2.2a).

Time Decoupling. Publishers and subscribers do not need to be participating in the interaction (with the event service) at the same time (Figure 2.2b). A publisher can publish events while a subscriber is disconnected, and a subscriber can be notified about events while the original publisher of the events is disconnected.

Synchronization Decoupling. The production and consumption of events happen in an asynchronous manner (Figure 2.2c). Publishers do not block while publishing events, and subscribers can be asynchronously notified about events (while performing some concurrent action). This decoupling dimension is, sometimes, called flow decoupling.

2.2.2 Subscription Models

Usually, subscribers are interested in particular events or event classes, and not in all events. To specify the events of interest, several subscription models have been proposed. Next, we present the main P/S variants, namely *topic-based*, *content-based*, and *type-based*.

2.2.2.1 Topic-Based P/S

This subscription model, also known as subject-based P/S, uses *topics* or *subjects* to classify events [187]. Publishers publish events associated with a topic, while subscribers subscribe their interest in receiving events of a certain topic. This notion is very similar to groups in group communication systems [34, 53, 204]. The event space is divided into topics, corresponding to groups or logical channels. Thus, subscribing to topic `A` can be seen as joining group `A`, and publishing an event on topic `A` can be seen as broadcasting that event to all the members of group `A`. Basically, it maps individual topics to distinct (many-to-many) logical communication channels.

The topic names used in these P/S systems are usually quite static, i.e., they are specified as initialization arguments to the event service [85, 155]. Thus, the topics that exist in the system are either out-of-band information (and must be known a priori by the clients), or are dynamically discovered using some additional support given by the system (e.g., having control topic channels, where new topics are advertised).

A topic is simply a keyword (i.e., a string) that represents a name according to which events are classified. Its namespace can be flat [45, 296] or hierarchical [187]. With a flat namespace, topics only allow to represent disjoint event spaces. Whereas, with a hierarchical namespace, topics can be structured according to containment relations. A subscription made to some topic in the hierarchy implicitly entails subscriptions to all its subtopics. Usually, these hierarchical topic names have a representation similar to URLs or file system paths. For instance, the topic `/StockMarket/StockQuotes` allows to subscribe for all stock quote events in a stock market. Whereas, `/StockMarket/StockQuotes/Telco` specifies the interest in being notified about stock quote events concerning only telecommunication companies. Some P/S systems also allow topic names to contain wildcards [187, 270].

In this subscription model, topics are the only information required to make events get from publishers to subscribers and, at the same time, topics are an *integral* part of the events. However, the event service only knows how to interpret topics and not the rest of the events' content, which remains opaque to the service.

Sometimes, the earlier term *channel-based* P/S is used to refer to a flat topic model where the topic name is not explicitly a part of the event, but of the channel to which subscribers register to [103, 113]. Publishers publish events to the channel, and subscribers connect to the channel and listen to all the published content.

Due to its simplicity, this model can be efficiently implemented. However, despite the hierarchical topic namespace and wildcards, this simplicity may imply a rather static model and limited expressiveness.

2.2.2.2 Content-Based P/S

The content-based P/S variant presents a more flexible subscription model based on the actual content of the events [5, 43]. Contrary to topic-based systems, in this subscription model, the events' content is interpreted by the event service for matching them against subscriptions. In this subscription model, the events' content is usually modeled as a set of attribute-value pairs.

Events are classified by their properties: either internal attributes of the events' content [19, 42, 64, 238], or metadata associated with the events [113]. Subscribers register their interests by specifying constraints over the values of the events' attributes using a subscription language (e.g., SQL, XPath, or some proprietary language). A subscription can be seen as a boolean function over predicates (e.g., `Company` == "Telco" `AND 20` < `Price` < 50). An event matches a subscription if the boolean function evaluates to true.

The event's attributes (and every kind of content to which clients can subscribe to), i.e., the event schema, have to be predefined. This schema is either out-of-band information known by all clients, or it must be dynamically discovered using some additional support from the system.

When compared to the topic-based subscription model, this one allows a more fine-grained filtering of events. Nonetheless, this increase in expressiveness comes at the cost of increased system complexity. In this subscription model, the main challenge is to efficiently match events against subscriptions without computing several repeated subscription evaluations.

Some examples of P/S systems employing the content-based subscription model are Elvin [238], Siena [42], JEDI [64], Hermes [203], and IBM's Gryphon [19].

2.2.2.3 Type-Based P/S

In this subscription model, events are filtered according to their type [84, 87, 88], i.e., events are actually objects of application-defined types (which can encapsulate attributes as well as methods). In turn, subscribers register their interest in receiving events of a specified type (and its subtypes). Thus, matching between events and subscriptions is transformed into type checking.

This subscription model presents an object-oriented approach, enabling a close integration of the programming language with the P/S system. By ensuring type safety at compile-time, this model includes events and subscriptions as first class citizens into the programming language.

Comparing with the topic-based subscription model, the event type is the topic. However, a type is more general than a topic and they may also support operations that can model content-based filtering (e.g., through predicate verification using the public members of the considered event types).

Table 2.2: Comparison of subscription models [155].

Model	Filtering	Publication	Subscription
Channel-based	No filtering	Events	Listening to channels
Topic-based	Topic (hierarchy)	Events tagged w/ topics	Topics (w/ wildcards)
Content-based	Event content	Events	Content-based filters
Type-based	Type checking	Objects	Object types

2.2.2.4 Comparison

In sum, each P/S subscription model offers a different degree of expressiveness, as well as distinct system complexity and performance overheads (naturally, depending on their specific implementations).

Topic-based P/S presents a simple model whereby a single name alone can determine all the relevant event recipients. It is due to this simplicity that exist many efficient implementations of this model. However, despite the hierarchical topic namespace and wildcards, this simplicity provides a rather static model and limited expressiveness.

On the other hand, content-based P/S provides a more flexible and general subscription model with improved expressiveness. Still, the additional expressiveness of this model comes at a price. Since event recipients can only be determined after each publication, it requires complex protocols possibly demanding higher runtime overheads, thus being more difficult to implement efficiently.

In turn, type-based P/S sits somewhere in the middle of the previous models. It provides a coarse-grained structure to events, like in topic-based systems, but also allowing fine-grained constraints to be expressed over the events' attributes (like in content-based systems) or over methods.

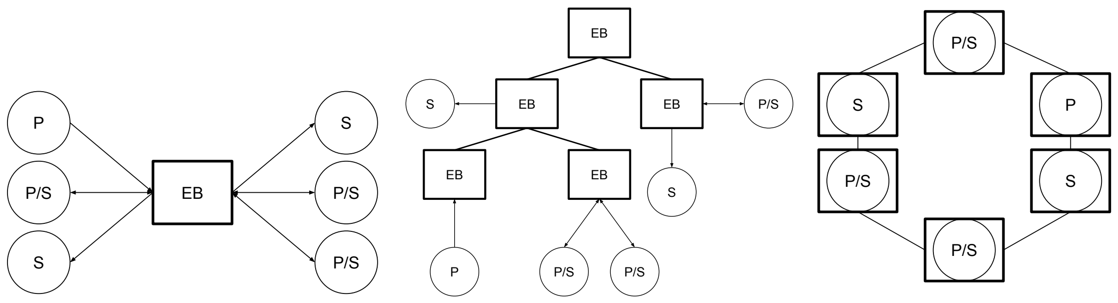
Table 2.2 sums everything up, comparing all the presented subscriptions models (including also the channel-based model) regarding filtering, publication, and the subscription capabilities each one supports.

2.2.3 System Architecture

The P/S system architecture concerns the way the interacting participants are organized, and with whom and how they communicate. Basically, it determines who does the matching and routing of events in the system. Matching determines who are the recipients of an event, while routing delivers an event to all its relevant recipients.

On the one hand, these tasks can be performed by event brokers, i.e., dedicated servers of the system where clients connect to. These servers are also responsible for executing the complex protocols required for persistence, reliability, or high availability.

On the other hand, these tasks can also be done by the clients themselves, i.e., a symmetric (P2P-like) system where participants share the same roles—they can either be a publisher, a subscriber, an event broker, or any combination thereof. The main



(a) Centralized. (b) Distributed. (c) Decentralized.

Figure 2.3: Examples of publish/subscribe system architectures (EB - event broker, P - publisher, S - subscriber, P/S - publisher and subscriber).

alternatives, which are depicted in Figure 2.3, can be classified as centralized, distributed, or decentralized approaches.

2.2.3.1 Centralized

In a centralized P/S system, the event service consists of a single broker to which both publishers and subscribers connect to. Figure 2.3a illustrates this star topology where the (central) event broker stores and manages all subscriptions, matches incoming events against the stored subscriptions, and then routes events to the matching subscribers.

The Java Message Service [113] provides a model that uses an event broker that is conceptually centralized. Since there is a central authority coordinating all the interactions in the system, the three P/S decoupling dimensions can be trivially ensured. However, the event broker presents itself as a single point of failure and as a bottleneck, thus this approach does not scale very well (and is not widely adopted in practice).

2.2.3.2 Distributed

In this type of P/S system, the event service is implemented as a distributed network of brokers, i.e., there are multiple event brokers to which publishers and subscribers can connect to. Usually, these broker networks follow some topology, such as a hierarchical structure (as a tree) [19, 64] or a general graph (with some constraints) [42, 203].

The tree topology presents a hierarchical relation among event brokers. Usually, clients can connect to any broker (similar to Figure 2.3b). This hierarchical structure is designed for scalability. A parent broker will receive events and subscriptions from all the clients connect to it, but will only forward down the events intended for its sub-tree. However, topmost brokers tend to be overloaded, and the failure of a broker might disconnect the entire sub-tree.

In the case of the general graph approach, systems usually tend to construct graphs following some routing protocol, and constraining connections taking into account the subscriptions registered in the system and their relations. Basically, they construct routing tables based on the registered subscriptions and route events accordingly [42].

2.2.3.3 Decentralized

In decentralized P/S systems, the event notification service follows a P2P approach where clients are the event brokers, i.e., there are no dedicated servers. Thus, in this approach, all participants play the role of brokers, matching and routing events. As the previous architecture, brokers usually adopt some common topologies, like trees, rings [45], or general graphs. For instance, Figure 2.3c presents an example of a set of brokers/clients structured as a ring.

Additionally, there are other systems that use no brokers, and allow publishers and subscribers to interact directly among each other [22, 270]. As such, these systems do not ensure the adequate P/S decoupling, but they can be suited for fast and efficient delivery of transient data.

2.2.4 Event Routing

As already mentioned, event routing is the act of delivering events to all their relevant recipients. This can be done by dedicated event brokers or by the clients themselves, depending on the system architecture (§2.2.3).

Common to all event routing approaches is the need to disseminate some piece of information from publishers and/or subscribers to the event brokers. The trivial solution for this task consists in disseminating each event or each subscription to all the brokers. Thus, allowing them to make local decisions. The natural drawback of these solutions is that they do not scale in large-scale or highly dynamic systems. In the middle sit several routing approaches based on selective, or probabilistic event routing.

2.2.4.1 Event Flooding

This is an extreme solution, that sits in one end of the spectrum of event routing approaches. Following this approach, events are flooded to all the brokers, i.e., to the entire system. Thus, after a publication, a publisher broadcasts the event, and filtering is done on the receiving side by the interested parties. Each client stores its subscriptions locally and upon receiving a notification, it checks if the event matches any of its locally registered subscriptions. If not, the event is simply filtered out.

This approach is very straightforward to implement, but very expensive. Its main inconvenience is its high message overhead, because events are disseminated to all brokers, whether or not they are serving any interested parties. However, it presents a minimal memory overhead, because only local subscriptions have to be stored.

2.2.4.2 Subscription Flooding

On the other end of the spectrum of event routing approaches lies the subscription flooding approach. This solution follows the opposite idea, whereby subscriptions are flooded to all the system brokers (instead of events) [39, 238]. Thus, each broker gathers the complete knowledge of all the system's subscriptions, and can then build a completely local subscription table. In the end, these tables are used to locally match events and directly notify the interested subscribers (and non-interesting events can be immediately filtered out at the publishers).

This approach is also very straightforward to implement, but it suffers from a large memory overhead, because every broker needs to store all the subscriptions registered in the entire system. However, event dissemination can be optimal, since events can be routed only to the brokers of interested subscribers. This approach becomes impractical for scenarios where subscriptions can change frequently.

2.2.4.3 Filtering-Based Routing

One approach that follows selective event routing is filtering-based routing. In this approach, subscriptions are partially disseminated in the system (following some rules), and are used to build routing tables. These routing tables are then exploited during event routing to dynamically build paths (e.g., multicast trees) that connect the publisher to the interested subscribers.

Due to this event filtering approach, events are forwarded only to nodes that lie on a path leading to interested subscribers. Thus, message overhead is reduced by identifying, as soon as possible, events that are not interesting for any subscriber ahead and stop their forwarding. Additionally, here, subscription dissemination can be restricted by exploiting containment relations among subscriptions [42].

2.2.4.4 Rendezvous-Based Routing

The other type of selective event routing is rendezvous-based routing. This solution is based on two functions, SB and EB, used to associate respectively subscriptions and events to specific brokers in the system. Given a subscription s, $SB(s)$ returns the set of brokers (named rendezvous nodes) responsible for storing s and forwarding received events matching s to all the relevant subscribers. Given an event e, $EB(e)$ returns the set of brokers that must receive e to match it against the subscriptions they store.

In this approach, event routing is a two-phase process. First, an event e is forwarded to all brokers returned by $EB(e)$, then those brokers match it against the subscriptions they store and notify the corresponding subscribers. For this approach to work, the mapping intersection rule must hold: $\forall s, \forall e : e.matches(s) \implies EB(e) \cap SB(s) \neq \emptyset$.

This approach tries to achieve better load balance for subscription storage and management. Since all subscriptions matching the same events are stored in the same (usually small) set of brokers, it avoids a redundant matching to be performed in several different brokers. Event delivery can also be simplified, for instance, by creating diffusion trees starting in the target brokers and spanning all the relevant subscribers.

Nevertheless, defining functions SB and EB to satisfy the mapping intersection rule may be far from a trivial task (for instance, when addressing the multi-dimensional nature of content-based P/S systems).

Examples of systems from the literature that use this event routing approach are Scribe [45], Hermes [203], Meghdoot [106], or Bayeux [296].

2.2.4.5 Probabilistic Routing

Probabilistic event routing follows the ideas of gossip-based protocols. This is a fully distributed approach, where each node contacts some of its neighbors (chosen at random) in each round, and exchanges information with them. The flow of information resembles the spread of an epidemic and leads to high robustness and reliability in highly dynamic networks. Being randomized, these protocols are simple and do not require nodes to maintain routing data structures.

The random choice of the neighbors to contact can also be driven by local information (e.g., the state of the subscriptions distribution in the network), and follow a similar principle to filtering-based routing: avoid to gossip events to non-interested subscribers [86]. This is sometimes called informed gossip.

Its main drawback is a moderate (and usually configurable) redundancy in message overhead compared to deterministic solutions.

2.2.5 Mobile and Wireless Environments

The first P/S systems were developed for completely wired environments, thought for supporting distributed applications built on top of static and managed environments. Thus, both event broker(s) and clients were wired devices connected through some network. Because of their design and underlying materialization, these systems are very rigid and have a rather static but stable topology.

However, in the last decades, computing devices started to become more and more mobile, and the environments where they operate are largely unmanaged. Accompanying that trend, P/S systems had to adapt and started supporting mobile clients, although event brokers continued to be materialized by (networks of) dedicated servers attached to a network infrastructure. With this design, broker networks still present a static topology, while clients are allowed to move (either logically or physically [93]). These systems assume clients have access to the network infrastructure (in order to access the broker network) and address clients roaming among different brokers through various techniques, like client proxies [40], mobility prediction [93], location-dependent subscriptions [92], or requiring clients to explicitly inform they are moving/connecting to another broker [65].

Although these systems address client mobility, event brokers are assumed to be wired dedicated servers. Thus, they are static wired devices that reside inside a network. However, because of the massive adoption of mobile wireless devices nowadays, there are some scenarios where the P/S paradigm might be of use but where there may not be access to a network infrastructure, such as locations without network access or with low/intermittent connectivity. Here, the completely decoupled interaction model of P/S perfectly suits the interoperability needs of such wireless and infrastructure-less scenarios (e.g., aids mobility and disconnected operations, and multicast delivery can exploit the intrinsic broadcast properties of the wireless medium). Therefore, several P/S systems were designed for WANETs, where there are no APs and system-wide services, such as STEAM [172], Mires [253], Fadip [196], or GeoRendezvous [41]. STEAM is a middleware service where subscribers only consume events produced by geographically close-by publishers (relying on proximity-based group communication). Mires was developed targeting WSNs. Fadip uses an unstructured approach based on gossip techniques. GeoRendezvous is based on a geographic hash table (GHT) for wireless networks, and uses its geographic properties to reduce notification latency.

In conclusion, regarding the environments event brokers and clients operate in, we can divide P/S systems in three different categories: 1) purely wired (both event broker and clients are wired, e.g., a P/S system for a local wired network); 2) partially wireless (a wired event broker and wireless clients, e.g., a P/S system for a WLAN); and 3) purely wireless (both event broker and clients are wireless, e.g., a P/S system for a WANET).

2.2.6 State Persistence

In most P/S systems, events are transient, i.e., once matched and disseminated they are not further stored or processed. Thus, only subscribers online at the time of publication will be notified about an event. Although many P/S systems addressing mobility allow subscribers to get undelivered events (e.g., due to a disconnection) when they reconnect, they do not allow new subscribers to explicitly request those past events.

In highly dynamic and volatile scenarios, or with frequent disconnections, the retrieval of past publications might be a necessary feature. In fact, this feature is the time decoupling of the P/S paradigm [85]—one of its decoupling dimensions. However, many systems require publishers and subscribers to participate in the interaction with the event service at the same time, in order to be notified of current publications. Exceptions are *state-based* P/S systems [155] and the *subject space* model [151, 152], whereby events persist in the system (for some time) after their publication.

Much work has already been done in P/S systems, both for wired and wireless settings. However, the notion of persistence (or time) has not been addressed in most. The subject space model [151, 152] formalizes P/S-style interactions and generalizes the P/S concept. It proposes a system that treats the relations between publications and subscriptions as a kind of state machine, maintaining the state between each pair of publication and subscription, and only sending notifications upon state transition (i.e., when the state of subscriptions change from false to true).

In environments with *wired* broker networks (although clients can be wireless and/or mobile), some systems explore the concept of a persistent data repository. This is achieved through distributed buffers [55, 90, 120, 251, 261] or caches [252] in the broker network, with a centralized component [33], or by integrating the P/S system with databases [153, 205, 239, 248, 277]. For instance, some systems use proxy servers that maintain permanent connections to the broker network, and buffer any notifications received while clients are disconnected. A drawback is that clients need to reconnect to the same proxies to receive the buffered notifications.

Regarding *purely wireless* settings, to the best of our knowledge, Chapar [137] is the only state-based P/S system. It targets MANET environments, and uses a broker network based on an OLSR overlay (with its known overheads) to handle publications and subscriptions. It buffers notifications in replicated data containers until they expire or they are delivered to all their intended subscribers. Due to its design choices, it presents some limitations. Since the system is not symmetric, broker nodes will have to do more work and spend more of their resources (e.g., storage, computing, battery) when compared to non-brokers. It employs both a subscription and event flooding approaches (the latter only for events that should be persistent), known for not scaling in highly dynamic and large-scale systems.

2.2.7 Complex Event Processing

Despite the plethora of features provided by P/S systems, they do not allow to express queries spanning multiple events. In turn, complex event processing (CEP) systems process incoming information as streams of events and work to detect certain patterns in the flow of events [161]. Large numbers of events that in isolation are not useful, need to be correlated to detect higher-level (composite) meaningful events [155].

The main goal of CEP systems is on detecting occurrences of particular patterns of (low-level) events (that can be filtered, combined and transformed), indicating some higher-level events. Contrasting to traditional P/S systems, CEP systems allow consumers to express their interests not in individual events, but in composite events, i.e., combinations of multiple events correlated over time. Additionally, some CEP systems also address time relations between events [25, 66].

2.3 Livin' on the Edge: Data Management at the Network Edge

Besides the obvious drawback of being a single point of failure and a bottleneck, centralized solutions for data management also demand significant monetary and operational costs, and overheads in its development, deployment, and maintenance. In turn, a decentralized solution spreads those concerns among several entities. If those entities can work collaboratively, such costs and overheads can be greatly reduced, or even eliminated.

Data management in our target environments involves several aspects, from which we highlight data storage—where and how data is stored—and data dissemination—how data is searched and retrieved, or how data is propagated to all interested stakeholders. Next, we present an overview of the state of the art research addressing these topics, ranging from wired to edge/fog computing environments.

2.3.1 Data Storage

As already mentioned, here, the data storage aspects that we explore concern about how and where data is stored in a specific system, and what kind of guarantees are provided to the users.

Krowd [79] and Ephesus [243] both enable generic content storage and sharing among nearby mobile devices. They provide a key-value store abstraction for networks of co-located devices, without relying on any centralized component. While Krowd relies on a one-hop DHT, requiring each device to know and connect to every other device in the network, Ephesus is sustained by a classical DHT, requiring only partial knowledge of the network. The authors of Krowd decided not to handle churn implicitly because it can consume too much bandwidth (a solution given by the authors is to re-issue key-value pairs periodically). Thus, it does not address device mobility or failure, nor data availability. In turn, Ephesus leverages its DHT to tolerate churn, and uses a popularity-aware replication mechanism to address data persistence and availability. Both require some kind of network infrastructure in order for devices to communicate among each other (e.g., an AP, or a device as a Wi-Fi hot-spot).

MobiTribe [268] is a system for content sharing on mobile devices, across the Internet. It uses a central server for content discovery, peer registration, and metadata management. Data is replicated in several peers according to their interests, and it uses intricate pre-fetching techniques to improve retrieval latency.

PAN [162] and Phoenix [195] are two systems for reliable storage in MANETs, providing similar mechanisms for data replication in ad-hoc networks. For that, PAN uses a protocol based on probabilistic quorum systems, and Phoenix uses a round-based simple quorum protocol (for one-hop networks). Both protocols allow the ingress/egress of peers to/from the network, and provide some guarantees regarding data persistence and availability. However, while PAN supports both concurrent access and update operations on stored data, Phoenix only allows access to its stored data. Still, because PAN is an asymmetric system, some peers will perform more work than others, causing imbalance in spent resources. In turn, being Phoenix a symmetric system, will probably provide for fairness concerning the resources spent by its participating peers.

Tuple spaces [7, 98] is another interaction paradigm that was adapted for mobile and wireless environments (§2.5.2). It provides a shared repository of immutable structured information, called tuples, that can be manipulated concurrently. It can be seen as an approach identical to distributed shared memory. Similar to the P/S paradigm, it also provides decoupling in time and space.

TierStore [73] is a distributed file system for challenged networks, such as delay tolerant networks (DTNs). It uses the DTN's store-and-forward routing strategy and a P/S multicast replication protocol to provide a standard file system interface. It implements an easy-to-reason-with conflict management policy, providing automatic conflict resolution in some cases.

Regarding storage solutions leveraging on edge computing capabilities, several approaches have been proposed. Some propose to build P2P overlays for storage, but using edge devices as peers in the overlay, i.e., storage clouds using edge devices [29, 185]. Thus, laptops, media centers, set-top-boxes, modems, or mobile devices (and even the users' cloud storage accounts), all contribute with storage space to create a single unified storage system, where data is cached, replicated, and placed to enable reliable access while minimizing latency and storage costs. Combining all end-user devices may result in a scalable and flexible storage cloud that keeps data close to the users (similar to content distribution networks (CDNs)), increasing availability while reducing latency.

FogStore [107, 171] goes even further and provides mechanisms to enable typical made-for-cloud distributed storage systems (such as Apache Cassandra) to operate transparently in edge/fog computing environments. Its focus is on data consistency, offering differential consistency guarantees for clients based on their context. Contrary to our target scenarios, FogStore addresses highly geo-distributed scenarios.

Curiously, we found several systems named EdgeStore (from different authors) [112, 138]. They have in common the fact that they leverage on edge servers to deploy such storage systems. Then, they use different techniques (such as pre-fetching, or client movement prediction) to improve read performance and user experience, as mobile clients move between edge replicas.

Similarly, PathStore [182], and its evolution SessionStore [181], provide a hierarchical store supporting eventual and session consistency, respectively. It consists of a persistent replica at its root (i.e., in the cloud), and an unlimited number of layers of partial replicas below it (i.e., in edge nodes). Clients are free to connect to any replica.

Other approaches devise several techniques to cache data close to the end-users, leveraging on storage resources available at the network edge (like wireless APs or mobile devices themselves). Some proposals use popularity-based caching and/or pre-fetching techniques to capture both long- and short-term content access patterns [269, 292], exploiting the transient co-location of devices, the epidemic nature of content popularity, and the capabilities of smart mobile devices. While some proposals store generic data [207], others provide more specialized features (e.g., Cachier [77] for computer vision applications).

In [274], the authors present a very insightful study of different edge applications, their data sharing needs, and the designs of some state-of-the-art systems (ranging from machine learning to gaming and autonomous driving). Nonetheless, all these storage services work from the edge (and even the cloud) down, deploying daemons on edge nodes/servers, and having mobile devices just as clients issuing requests to servers.

2.3.2 Data Dissemination

In turn, the data dissemination aspect can be seen as how data is searched and retrieved, or how data is propagated to all the interested stakeholders in a specific system.

iTrust [157] and CoQUOS [209] both propose lightweight probabilistic approaches to data search and retrieval. However, while iTrust has an implementation for WANETs comprised by Wi-Fi Direct groups, CoQUOS' target environment is typical unstructured overlays in the wider Internet. The two systems use random walk techniques for propagating both content advertisements/metadata and queries through the network. In CoQUOS, queries are installed only in certain peers following some probabilities that depend on the number of hops, while in iTrust, metadata is stored in every node traversed by the random walk. While CoQUOS follows a more reactive interaction model where queries are installed in the network and new content notifications are sent to the interested peers, iTrust follows the request/reply model, where peers have to proactively search for content. Their main drawback is data persistence and availability, i.e., when a peer leaves the network, its published content becomes inaccessible.

Information-centric networking (ICN) [6] departs from the host-centric network architecture, where there is constant end-to-end connectivity, to a data-centric architecture, where content is directly addressable by its name. We introduce this concept in §2.5.1.

Peer data sharing (PDS) [249, 250] is another system that provides content search and retrieval for networks of mobile devices. It follows an approach inspired by ICN, adopting a content-centric design where (immutable) data is referenced and accessed by name, independently from where it was produced or resides. Due to its ICN approach, data can be widely cached throughout the network, and thus retrieved from any willing and capable peer. It targets small scale networks with low to moderate mobility, and it adopts a proactive request/reply interaction model to discover what data exist in nearby peers. Because of its aggressive caching policy, it can lead to serious storage overheads, and since data is only cached (and thus replicated) if peers request it, its data availability and persistence characteristics are not that strong. As a result, only popular items have some availability guarantees, while less popular data may even disappear.

Theia [226] and MediaScope [127] present systems for content search in mobile devices. Both consider smartphones as "distributed databases" and allow users to compose queries on a centralized server and push them onto the registered smartphones to find out photos that match the specified queries.

Both Haggle [186] and PodNet [170] provide communication and content sharing in the presence of intermittent connectivity, i.e., in opportunistic and delay-tolerant networks. They depart from the traditional end-to-end communication abstractions, eliminating many of the network layers above the link layer, and adopting a strategy more oriented to the human way of communicating—through communities and their interactions. They exploit opportunistic contacts between mobile users to deliver data to the destination. Their main difference resides in the fact that while PodNet provides content sharing using topic- and pull-based dissemination, Haggle follows a strategy more in line with content-based P/S systems, also providing ranking of matching content. ContentPlace [36] also provides content dissemination for opportunistic networks. It is very similar to PodNet, but it exploits information about users' social relationships to decide where to cache data and how to propagate it.

Floating Content [191] and 7DS [176] are other two data dissemination approaches for DTN environments. They take advantage of opportunistic contacts between peers to allow the exchange of information, spreading it through the network, but with time and space limitations. Of course, this means content dissemination is best-effort, i.e., the information spreading depends of the availability and willingness of interested nodes to carry such content. In [194], the authors explore the Floating Content approach using WiFi Direct in Android mobile devices. Similarly, BitHoc [142] explores an identical idea using a tracker-less BitTorrent-like application for wireless ad-hoc networks.

2.4 Put a Ring on It: A Review on Distributed Hash Tables

P2P computing appeared as an alternative to the typical client-server architecture where clients request services and resources from centralized servers [234]. Instead, peers (or nodes) have equal roles and form a network in order to share their resources among each other without resorting to centralized entities. Thus, peers have the capability of acting as servers and as clients at the same time. The music-sharing application Napster [229] was probably the first to popularize the concept of P2P.

P2P networks implement some form of (virtual) overlay network on top of the physical network topology. Thus, at the application level, peers communicate directly among each other (via the logical overlay links). Taking into account how peers are connected to each other, these overlay networks can be classified as *unstructured* or *structured* (or also as a hybrid between the other two categories) [131].

2.4.1 Overlay Networks

An overlay network is defined as a network which is layered on top of another network [160]. Nodes in the overlay network can be seen as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links in the underlying network. The links that compose an overlay network are named logical as they are independent of the underlying network links and topology. This means that two direct neighbors in the overlay, may be separated by several hops in the underlay, and vice versa. Here, we are only going to refer to overlay networks that operate at the application level.

Peers form an overlay network by connecting among each other (forming some kind of graph). These connections, or neighboring relations, are then captured as neighbor sets managed by each peer. Typically, these neighbor sets are maintained by some kind of distributed membership protocol, which is in charge of dealing with filiation dynamics. Managing these neighbor sets can be a complex task, in particular because these overlays' filiation may be subject to churn—the (fast paced) independent arrival and departure of several peers, as well as their failures [258].

As mentioned, overlay networks can be divided into two main categories by taking into consideration the mechanisms used to maintain the neighbor sets.

Structured overlay network [256]**:** an overlay network which relies on a global coordination mechanism, based on unique peer identifiers (e.g., an overlay organized in a ring, ordering peers according to their identifiers). Such schemes allow to deterministically infer the location of a peer (i.e., its neighbors) in the overlay given the identifiers of other peers in the system.

Unstructured overlay network [229]**:** an overlay network that has a random topology, such that it is impossible to predict where a node will be positioned. These overlays have a large degree of freedom when managing their topology in the presence of changes in the system (e.g., churn).

In the thesis, we address structured overlays in more depth, as such, we delve further into that topic in the next section.

2.4.2 Structured Overlay Networks & DHTs

The most common example of a structured overlay network is a DHT [211, 221, 256]. Other different examples are skip graphs (such as SkipNet [114]).

In DHTs, both peer ids and keys are mapped to the same key space/domain. In turn, the system maps any given key to a peer which identifier is the same, or the closest to the given key. Their popularity turned them into an important building block in many distributed systems. They provide a scalable lookup service, used to build complex large-scale systems, such as distributed storage [145], or web caching [94].

Peers self-organize into a specific network topology (or overlay) to provide key-based routing (KBR), efficiently mapping a given key onto a peer in the overlay, called the *key owner*. All this is done through a single operation: `route(message, key)`—route a given message to the peer responsible for a given key. Additionally, DHTs provide this feature while limiting the size of the neighbor sets each peer is required to maintain.

On top of KBR, DHTs usually implement data storage by associating a value (i.e., a data item) with each key, and storing the key-value pair at the key owner [68]. With this, like it sounds, a distributed hash table provides the same typical operations offered by its non-distributed counterpart: `put(K, V)` and `get(K): V`.

Hashing [133] is used as the key space partitioning scheme, i.e., to assign ownership of a key range to a specific peer, uniformly spreading keys among peers, and thus achieving a balanced load across the overlay.

DHTs have been widely studied, leading to the proliferation of several designs [105]. Some present a ring structure (e.g., Chord [256]), while others present a tree-like structure (e.g., Pastry [221], Tapestry [295]), or even a hyper-cube structure (e.g., CAN [211]).

In the thesis, we address specifically ring DHTs, as such, we delve further into that topic in the next section.

Table 2.3: Definition of variables for peer p, using m-bit identifiers [256].

Notation	Definition
$finger[k].start$	$(p+2^{k-1}) \bmod 2^m, 1 \leq k \leq m$
$finger[k].interval$	$[finger[k].start, finger[k+1].start)$
$finger[k].peer$	first peer $\geq p.finger[k].start$
$successor$	the next peer on the identifier circle; $finger[1].peer$
$predecessor$	the previous peer on the identifier circle

2.4.2.1 Ring DHTs

Ring-based DHTs, such as Chord [256], organize peers in a ring-like topology, and keys are placed in specific peers using consistent hashing. The ring is created by sorting peers using their identifiers (modulus the size of the key space).

In this section, we explain several concepts using the definitions and remote procedure call (RPC) notation of the Chord paper, as shown in Table 2.3. Thus, p.FOO() denotes an RPC of procedure FOO on peer p, while $p.bar$ (without parentheses) is an RPC to fetch attribute bar from peer p.

The overlay's maintenance is mostly proactive. Each peer keeps up-to-date pointers to its predecessor and successor peers through a periodic maintenance procedure (also called a *stabilization* procedure), depicted in Algorithm 2.1. More specifically, each peer r periodically asks its successor s for the predecessor p of s. Naturally, in a stable scenario, $p = r$. Should $p \neq r$ be a peer with an identifier in the range (r_{id}, s_{id}), r will then update its successor pointer to this new peer p. After this update, r notifies p that it is now p's predecessor (through the NOTIFY procedure). This simple procedure allows the ring to converge and stay connected even in face of (concurrent) ingress and egress of peers.

Using only the ring pointers, a peer can reach any other in the overlay, albeit in a very inefficient way (e.g., forwarding a message around the ring until its destination). Thus, these overlays usually employ a more efficient routing mechanism, being able to route messages in $log(N)$ overlay hops, where N is the total number of peers. For this, each peer maintains a *finger table*, from which it selects the closest peer on the ring to route

Algorithm 2.1 Ring DHT maintenance procedure [256].

```
1:  procedure STABILIZE( )          ▷ periodically verify my immediate successor, and tell the successor about me
2:      $x \leftarrow successor.predecessor$
3:      if $x \in (me, successor)$ then
4:          $successor \leftarrow x$
5:      $successor$.NOTIFY($me$)

6:  procedure NOTIFY($n'$)                                        ▷ $n'$ thinks it might be my predecessor
7:      if $predecessor = \perp \vee n' \in (predecessor, me)$ then
8:          $predecessor \leftarrow n'$

9:  procedure FIXFINGERS( )                                       ▷ periodically refresh finger table entries
10:     $i \leftarrow$ random index $> 1$ into $finger[]$
11:     $finger[i].peer \leftarrow$ FINDSUCCESSOR($finger[i].start$)
```

Algorithm 2.2 Ring DHT routing [256].

```
function FINDSUCCESSOR($id$)
    $n' \leftarrow$ FINDPREDECESSOR($id$)
    return $n'.successor$

function FINDPREDECESSOR($id$)
    $n' \leftarrow me$
    while $id \notin (n', n'.successor]$ do
        $n' \leftarrow n'$.CLOSESTPRECEDINGFINGER($id$)
    return $n'$

function CLOSESTPRECEDINGFINGER($id$)
    for $i = m$ down to 1 do
        if $finger[i].peer \in (me, id)$ then
            return $finger[i].peer$
    return $me$

procedure ROUTE($msg, k$)                    ▷ route message $msg$ to owner of key $k$
    $dst \leftarrow$ FINDSUCCESSOR($k$)
    SEND($msg, dst$)
```

messages towards their destination. This finger table contains pointers to peers which are at exponentially increasing distances from the peer's position in the ring, and also needs to be periodically updated (by procedure FIXFINGERS in Algorithm 2.1).

When a peer wants to route a message to a given key, it probably will not know the corresponding key owner, as peers only have a partial knowledge of the overlay topology. Thus, to do it in an efficient way, the peer leverages on its finger table and triggers a call to the FINDSUCCESSOR function, in Algorithm 2.2. Peer p searches its finger table for peer j whose id most immediately precedes key k, and asks peer j for the peer it knows whose id is closest to k. By repeating this process, p learns about peers with ids closer and closer to k. When executing the FINDPREDECESSOR function, a peer contacts a series of peers moving forward around the ring towards the given key. Eventually, peer p will reach the predecessor of key k, and then it only has to ask that peer for its successor—the key owner of k. Finally, peer p can contact the key owner of k directly. Similarly to domain name system (DNS) queries, this routing procedure can be executed in a recursive- or iterative-like manner (e.g., Algorithm 2.2 uses the iterative way).

In these overlays, the correctness criterion is to maintain a correct successor (since, in the worst case, we can route and make progress using only the successor pointers). Thus, for failure recovery, each peers keeps a *successor list* of its nearest successors in the ring, usually of size $log(N)$. If a peer notices that its successor has failed, it replaces it with the first live entry in its successor list. Throughout the overlay lifetime, the maintenance procedure also updates this list.

2.4.2.2 Load Balancing in DHTs

One of DHTs' fundamental issues is that peers or keys may not be uniformly distributed in the key space [144, 215, 263]. Therefore, some peers may become overloaded, having to store many keys or answer many queries, while others may be relatively idle.

There are several techniques to address query hot-spots. The use of virtual servers [68, 101] is one of them. With virtual servers, each peer joins the DHT using multiple identities, i.e., each physical peer maps to several virtual peers in the overlay. A large number of such servers can lead to an improved load balancing. However, it also implies that each peer manages more routing state and monitors more overlay neighbors, which may impose an excessive overhead (both in terms of storage and communication). Thus, the efficiency of this technique depends on how many virtual servers each peer can handle. Note that this mechanism magnifies the effects of churn, as the egress of one peer leads to the simultaneous departure of multiple virtual peers.

Other techniques tackle the same problem by making an informed decision about peer identifiers when they join the overlay [136, 147]. Locations in the key space are selected such that the load is evenly distributed among all peers, e.g., determining the best identifiers to use through probing. This technique trades (storage) load balancing for an increased cost when peers join the overlay, without requiring additional routing state. However, since identifiers are statically defined at join time, it may create a non-uniform distribution of peers in the key space, hampering both the storage load balancing and the performance of some routing algorithms over time.

Dynamic redistribution of keys tries to rebalance keys stored by peers [134, 210]. This is achieved either by moving more peers to overloaded parts of the key space, by moving virtual servers from overloaded peers to less loaded ones, or even by adding more virtual servers. However, this implies several bulky state transfers when moving keys, requiring lots of bandwidth. Additionally, this technique does not tackle all storage hot-spots, specifically the ones caused by popular keys having many associated values, since these are caused by a single DHT key. Also, by using virtual server-based techniques, we reap its benefits and issues (e.g., increase in the routing tables' size).

Caching keeps copies of previous queries in different peers to achieve better query performance and distribute the load of answering those queries [208, 255]. This can greatly reduce the number of query hot-spots. However, with caching comes always the problem of stale data and storage overhead.

2.4.2.3 Replication in DHTs

Several alternatives have been explored in the literature to support data replication in DHTs. They can be divided in three main categories [143]: neighbor replication, multi-publication, and path replication.

Neighbor replication keeps copies of each key in some neighbors of the key owner (e.g., its successors). It enables an easy and tight control on the replication degree, since when neighbors change, the key owner triggers the creation of new replicas to ensure that the replication degree does not fall below a target threshold. However, since peers join the overlay in a location that depends on their identifiers, and not on previous failures, they may not be able to quickly replace failed replicas. Furthermore, each node keeps a different set of replicas, which makes this bookkeeping extremely costly. Even more, if one attempts to use a flexible scheme where the number of replicas fluctuates (without ever going below a safety threshold). During its operations, this replication scheme creates replicas when peers fail, and moves them when peers join, which can become very costly [35]. Additionally, depending on the routing scheme used by the overlay, neighbor replication may not perform a fair load distribution, as some replicas are more likely to be hit by queries than others.

In turn, the multi-publication replication scheme stores a predefined number of replicas of each key in different locations of the DHT (e.g., using multiple hash functions, or by salting the key). Here, some mutual monitoring scheme is used to detect the departure or failure of replicas, and to subsequently restore them. On the one hand, it offers good load balancing properties, as multiple queries may be diverted to different parts of the DHT. However, monitoring can become extremely expensive, because it needs to use DHT routing and a key owner may be forced to monitor a different set of replicas for each key it owns. For instance, in the worst case, a peer that stores k keys, each replicated in p peers, has to periodically monitor $k \times (p-1)$ different peers. In practice, here, the bookkeeping problems of neighbor replication are amplified by the need to perform DHT routing during replica maintenance.

Lastly, the path replication scheme can be used for load balancing, and also to speed up queries. Following this scheme, replicas are created when queries are executed, by caching the corresponding results in the peers that forward the query back to the original requester. Since it is unfeasible to keep track of all replica locations, it is practically impossible to guarantee a minimum replication degree. Additionally, replicas can also be discarded in an uncoordinated manner.

Another approach to replication is group-based DHTs [108, 139, 192], where several physical peers are grouped together and work collaboratively as a virtual peer in the overlay. The peers inside a group all work to keep the same set of keys (and remaining state). In this approach, usually groups monitor their load, and react by splitting the overloaded groups and merging idle ones, redistributing keys among them.

Other solutions join (strongly consistent) consensus-based replication with DHTs. Etna [184] and Scatter [100] are key-value stores with support for data replication, that rely on Paxos[146] for maintaining a DHT ring composed of strongly consistent replication groups. However, while Scatter forms a DHT ring composed of (disjoint) replication groups, Etna allows different objects to be replicated on different sets of peers. They both rely heavily on consensus to implement every operation, therefore presenting high signaling costs and may block under heavy churn.

2.5 Potpourri: Other Relevant Topics

Here, we give a succinct presentation of some other topics that are also relevant to the thesis, namely information-centric networking and tuples spaces. These represent two substantially different ways of looking to data dissemination and search/query.

2.5.1 Information-Centric Networking

ICN is an evolution of the current Internet infrastructure [6]. It departs from the host-centric network architecture, where there is constant end-to-end connectivity, to a data-centric architecture, where content is directly addressable by its name, i.e., end-points communicate based on named data instead of IP addresses. So, it decouples data from its producer/source at the network layer.

Following this new paradigm, connectivity may be intermittent, data becomes independent from its location (thus in-network caching can be capitalized), and mobility is the norm. Figure 2.4 presents the mapping between how the Internet and ICN see the network. ICN tries to generalize that thin waist, in order to allow packets to refer (i.e., name) data objects instead of communication endpoints. By leveraging named data instead of named hosts, they do not require the maintenance of routing paths, making them well adapted to environments with intermittent connectivity and hostile propagation conditions [11]. It leverages heavily on several techniques such as in-network caching, multi-party communication through replication, and interaction models that decouple senders and receivers.

The ICN concept has been developed and evolved through several future Internet research projects. Even though their approaches to ICN differ in some details, they share many assumptions and properties. Some of these projects are data-oriented network architecture (DONA) [140], content-centric networking (CCN) [122], publish-subscribe internet routing paradigm (PSIRP) [75], named data networking (NDN) [294], content-based networking (CBN) [44], and network of information (NetInf) [69].

Although devised for wired environments, like the Internet, it was also adapted to mobile and wireless settings [11, 12, 49, 130]. Basically, it poses as a content dissemination

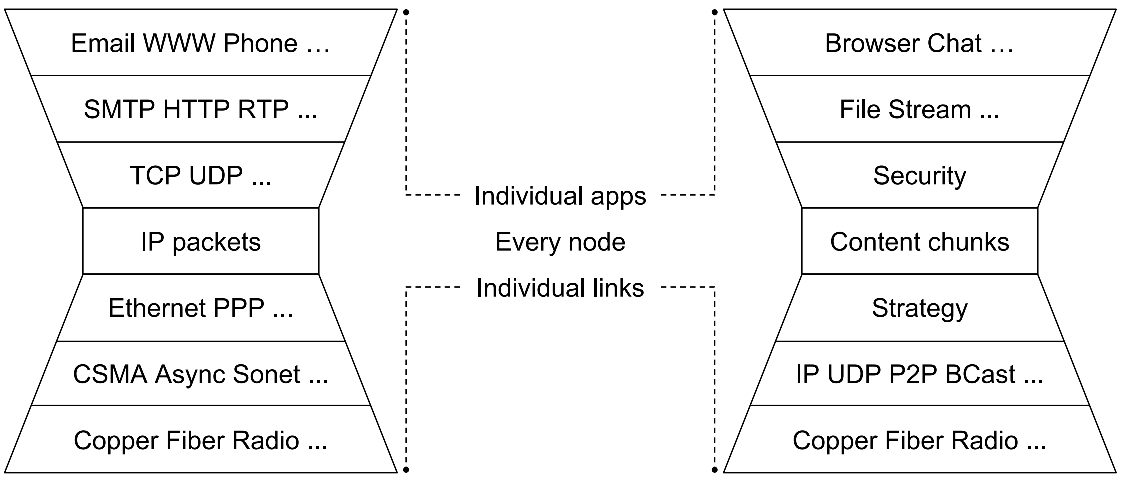


Figure 2.4: Internet (left) and ICN (right) hourglass architectures [44].

approach at the network level.

2.5.2 Tuple Spaces

The generative coordination model, originally introduced in the Linda programming language [7, 98], uses a shared memory abstraction called *tuple space* to provide coordination (and communication) between processes. This tuple space is basically a shared repository of immutable structured information, called *tuples*, that can be accessed concurrently. Tuples generated by processes are independent of the tuple space, thus any process may remove a tuple, and a tuple is bound to no process in particular. Similar to the P/S paradigm, it also provides decoupling in time and space.

A tuple is a sequence of fields, that can be left undefined. A tuple t with all fields having a defined value is called an entry. In turn, a tuple with one or more undefined fields is called a template (usually denoted by $\bar{t}$). Templates are used to allow content-addressable access to tuples in the tuple space. That is, an entry t matches a template $\bar{t}$ if they have the same number of fields and all defined field values of $\bar{t}$ are equal to the corresponding field values of t. For instance, template $\langle 2, *, 'd', * \rangle$ matches tuples with four fields, in which 2 and $'d'$ are the values of the first and third fields, respectively. Thus, a tuple space works as an associative memory, where tuples are accessed not by their address, but rather by their content.

A tuple space provides three basic operations: `in`($\bar{t}$) reads and removes a tuple that matches $\bar{t}$ from the tuple space; `rd`($\bar{t}$) reads a tuple matching $\bar{t}$ without removing it from the tuple space; and `out`(t) writes a tuple into the tuple space. Operations `in` and `rd` are blocking, i.e., they stay blocked until there is some matching tuple available. Most tuple spaces also provide non-blocking versions of these operations, `inp` and `rdp`. They work in the same way as their blocking counterparts, but if there is no matching tuple in the tuple space, some error code is returned.

These operations combined with the content-addressable capabilities provide a simple yet powerful programming model for distributed applications [98, 148]. Its main drawback is that the tuple space abstraction is based on an infrastructure, which is usually implemented as a centralized server, being a single point of failure.

Nonetheless, some works have addressed that main disadvantage in wired settings [31, 126], by implementing the tuple spaces over a distributed network of brokers, and also by using byzantine consensus. Even further, other works, like LIME [202], TinyLIME [67], TeenyLIME [62], TuCSoN [188], or TOTA [165], adapted the tuple space model for mobile and wireless environments. These are adaptations for environments ranging from MANETs to WSNs. One of their main differences is that these systems allow actions to be performed as *reactions* to certain events in the tuple space (e.g., inserting or removing a specific tuple). In order to adapt to those asynchronous environments, the Linda model was modified and extended. For instance, operations were transformed to be non-blocking and results started being returned through callbacks. Also, additionally to Linda's (proactive) operations to insert, read, and remove tuples, some allow for asynchronous notifications when relevant data appears in the tuple space. One drawback is that typically tuples are shared only among peers within radio range. In several mobile and wireless environments, this approach may not be sufficient to support the necessary distributed services or applications.

2.6 Concluding Remarks

In this chapter, we give the research context for the thesis by covering the most relevant technical background and state of the art. With this overview, we describe some topics related to our work that help to better understand this document. At the same time, we motivate the need for data storage and dissemination solutions for pervasive edge computing environments that, on the one hand, harness the capabilities provided by each level of the network hierarchy and, on the other hand, are able to operate in wireless ad-hoc settings.

Since we target wireless environments, we start by reviewing the existing types of wireless networks, and wireless ad-hoc routing protocols. Considering the asynchronous and highly dynamic environments we target, we also survey several important concepts around the best known loosely coupled interaction paradigm—the publish/subscribe paradigm. Next, we give an overview of the state of the art regarding data storage and dissemination in pervasive edge computing environments, reviewing approaches for different settings, ranging from wired to wireless and edge/fog computing environments. After that, we also discuss essential concepts regarding overlay networks and DHTs. Lastly, we give a brief presentation about some other relevant topics, such as tuple spaces and ICN.

Taking into account that we address data storage and dissemination solutions operating both in settings with or without network infrastructure access, we leverage on both infrastructure and infrastructure-less networks. When without access to some network infrastructure, our proposed solution makes use of multi-hop ad-hoc communication to allow devices to communicate among each other. From several studies [257], geographic routing protocols seem to be aligned to deliver the best trade-off between performance and expended resources, in larger networks.

These completely decentralized and highly dynamic environments we target require efficient and resilient ways of storing and searching for data. From our surveyed approaches, some use typical DHTs (although that adapted to a wireless environment). Others rely on some central entity for coordination, thus not suitable for this kind of environment. Still, others are asymmetric, providing unfairness in workload distribution, or designed for one-hop networks, thus limiting the applicable use cases. Others are devised for scenarios that tolerate delays of hour and even days (e.g., DTNs), making it impossible to allow for (near) real-time content sharing. Thus, the surveyed systems serve as inspiration for our proposal, but they also call for some new techniques able to provide the guarantees required in these environments.

Regarding data dissemination, without a central coordinator it is difficult for users to be aware of what data is being published or shared, thus they do not know what data can be retrieved. That is why we argue for a reactive interaction model, where users register their interests and are notified as new data is generated (and then decide if it is interesting enough to be retrieved). This approach also provides loose coupling between the interacting parties, a feature required in such volatile and dynamic environment.

When having access to network infrastructure, the storage service can leverage on existing edge computing capabilities and exploit the guarantees provided by each level of the network hierarchy. Like some of the solutions presented in §2.3, we harness the storage resources available at the network edge in order to provide a scalable and flexible storage solution keeping data close to the end-users. We also integrate the same loosely coupled reactive interaction model, i.e., the P/S paradigm, into the storage substrate, taking advantage of the existing edge servers to share management responsibilities.

In conclusion, although a few approaches address some of the challenges identified in §1.3, none focus on scenarios both with and without network infrastructure access, and do not integrate a reactive and loosely coupled data dissemination mechanism into the storage solution. Thus, from what we surveyed in this chapter, we conclude that our research questions, defined in §1.4.1, are interesting and still open.

Chapter 3

Time-Aware Reactive Storage

"Home is behind, the world ahead, and there are many paths to tread through shadows to the edge of night, until the stars are all alight."
— J. R. R. Tolkien

The thesis targets highly dynamic and asynchronous environments composed of co-located mobile devices. In these scenarios, devices are free to enter or leave, to move, and may also fail. Accordingly, we argue for a loosely coupled data dissemination mechanism. Thus, here, we present an in-depth definition of our data storage and dissemination model: time-aware reactive storage (TARS).

We introduce the addressed issue and give some context in §3.1. Next, in §3.2, we survey some related work and compare our approach with it. Then, in §3.3, we detail our TARS model. Lastly, we conclude the chapter in §3.4, by presenting a discussion of the resulting abstraction, and the publications emanated from this work.

3.1 Introduction

In typical storage systems [79, 162, 249], users are required to *actively* and *explicitly* search for the desired data, following a *request/reply* interaction model. However, as already mentioned in the previous chapters, the kind of distributed environments we target are highly volatile and dynamic, rendering this proactive interaction model really impractical, and possibly cumbersome to use due to disconnections. Accordingly, we adopt a *reactive* and *loosely coupled* data dissemination mechanism—the publish/subscribe (P/S) paradigm [85]. This simple communication abstraction provides decoupling in time, space, and synchronization between publishers and subscribers (§2.2), which facilitates loosely coupled, spontaneous interactions (required for this kind of dynamic and pervasive edge environments). By integrating a P/S abstraction, users (or applications) can register their interests, being subsequently notified of any data items matching those interests. This allows users to quickly discover what data exist in the system in a reactive manner, and *only* be notified about data they are interested in.

Additionally, in the kind of social gatherings we are addressing (e.g., sporting events, celebrations, concerts), individual moments are intrinsically tied by time relations, such as the band performing at time x in the music festival, or the second speech on a rally. Also, people are often interested in information with these associated time references (e.g., find photos of the opening band, or a video of the goal right before the match intermission). Therefore, we consider *time* to be a first order dimension. Thus, subscriptions comprise a time frame that defines their *active time-span*, which may include the future, the present, or the past, effectively providing the *full time decoupling* of the P/S paradigm.

With all this, we build strong synergies between the storage substrate and the P/S paradigm, and propose time-aware reactive storage (TARS), a reactive data storage and dissemination model with intrinsic time-awareness, allowing queries (i.e., subscriptions) within a specific time scope.

In summary, the main contribution of the work presented in this chapter is the definition of our time-aware reactive storage model, TARS.

3.2 Related Work

Concerning work related to this proposal, we address and compare against four main categories: P/S systems, tuple spaces, active databases, and continuous queries.

3.2.1 Publish/Subscribe

Typical P/S systems are stateless, meaning that only subscribers online at the time of publication are notified, i.e., consumers only receive data published by producers if online at the same time. Hence, the notion of publication persistence has not been addressed in most systems. Additionally, to the best of our knowledge, the notion of subscriptions with a time dimension (i.e., time-aware) is also not explored in related work.

Kafka [141], a system originated at LinkedIn, has recently gained significant popularity, clearly demonstrating the feasibility and potential of the P/S communication paradigm. However, such solutions do not consider time as a first order dimension of the P/S abstraction. Furthermore, solutions for wired scenarios cannot be easily adapted for wireless setting where connectivity is not stable and device population is volatile.

Table 3.1 highlights the main aspects of some P/S systems that address state persistence. Some approaches for wired settings exploit the concept of a persistent data repository, by means of distributed buffers [55], allowing only to specify how many data items to request from the past when subscribing. In turn, others propose their integration with traditional databases [277], but without any notion of time.

Table 3.1: Comparison of TARS with publish/subscribe systems.

	Environment	Time Assignment	Access Past
[55]	Wired	—	Num. Items
[277]	Wired	—	N/D
Chapar [137]	Wireless	Publications	Time-to-live (TTL) Pub.
TARS	—	Pub. & Sub.	Time-aware Sub.

In the particular context of wireless settings, Chapar [137] is, as far as we know, the only P/S system that addresses persistent publications. However, it only assigns time to publications, which are buffered only until their lifetime expires, as a TTL. In TARS, subscriptions have their time scope assigned. While publications are permanently stored, they may be deleted from the system upon request. Thus, new subscribers can always request previously published data. Moreover, Chapar is not functionally symmetric, demanding more work from broker nodes, and thus achieving poor load balancing. On the contrary, TARS' high level definition allows for different implementations (see §4).

3.2.2 Tuple Spaces

As already mentioned in §2.5.2, tuple spaces are an interaction paradigm for parallel and distributed computing, providing a shared data space abstraction. Also, some systems, like TuCSoN [188], LIME [202], and TOTA [165], adapted the tuple spaces model for mobile and wireless environments. Besides the model's proactive operations (for inserting, reading, and removing tuples), these systems allow actions to be performed as *reactions* to certain events.

Although reactions are similar to subscriptions in TARS, they have significant differences. First, reactions always execute on the client side, i.e., on the host that installed it, and always receive the tuple that triggered the reaction. This does not allow load balancing when executing the reactions and when matching reactions with tuples. Additionally, it also has the potential to generate more traffic than actually required, because it is not possible to filter data at the source. For instance, in our implementation of TARS (i.e., Thyme; see §4), subscription matching can be executed by randomly selected peers that may change in each matching (see §4.5), thus improving load balancing and optimizing the data to be delivered to each client.

Another major difference is that tuple spaces do not differentiate between data and metadata management, i.e., everything is represented as a tuple. In TARS, when receiving a notification, nodes only receive an object's metadata (containing a small amount of information), and only after that decide if the object is interesting enough and proceed to retrieve it. Since metadata is usually much smaller than the actual data, this strategy can considerably reduce network traffic. Additionally, when managing replication and mobility, metadata may require updates. Since tuples are immutable, the only way of modifying metadata is to remove and insert a new (changed) tuple, which may trigger unwanted reactions. This can be bypassed by making an intricate decomposition of the metadata into several (sub-)tuples. Although this may work in small scale scenarios, it can quickly become cumbersome, and penalize performance in large-scale scenarios, as targeted by our work.

Comparing with P/S, the latter is conceptually stateless, differentiating it from interaction paradigms like tuple spaces that are based on a shared data-space. However, the P/S infrastructure needs to maintain some state to support special features like disconnected operations. Here, by combining the two concepts in a certain way, TARS provides both data storage and dissemination through an asynchronous interaction model.

3.2.3 Active Databases

Traditional database systems are categorized as passive in the sense that operations are executed by the database (e.g., query, update, delete) as and when requested by the users or applications. On the other hand, active databases [197] move some of the reactive behavior from the application into the database. That is, they are able to monitor and react to specific events of relevance to an application. Typically they include an event-driven architecture, often in the form of event-condition-action (ECA) rules. The event part specifies the occurrence that triggers the rule. The condition part verifies the context in which the event was triggered. The action part consists of the task to be executed if the event was triggered and the condition is satisfied. Possible use cases include monitoring, alerting, and statistics gathering. Most modern relational databases include active database features in the form of database triggers.

This feature incorporates some reactivity into typical databases, making it very similar to our TARS approach. However, their major difference concerns time-awareness. Active databases, as do regular databases, allow the query of any attribute of a table (including time-related ones, if they exist). In turn, TARS elevates time as a first order dimension, and thus provides an intrinsic time-aware approach, where time does not have to be an integral part of the stored data items.

Usually, active databases assume a centralized environment. Consequently, the majority of its implementations do not consider distributed settings, dropping such feature in favor of others (such as performance).

3.2.4 Continuous Queries

A continuous query [20] is a query that is issued once over a database, and then logically runs continuously over the database until it is terminated. Thus, it lets users get new results from the database without having to issue the same query multiple times. They work contrary to traditional SQL queries, that run once to completion over the current data in the database and return the result to the user. In traditional databases, materialized views coupled with triggers can be seen as a kind of continuous queries. A materialized view is a query that needs to be reevaluated whenever specific data (over which the view is defined) changes. In turn, triggers allow the definition of ECA rules, enabling the database to take specific actions when certain events occur.

Whereas traditional databases manage data within some form of (persistent) data sets, in many recent applications where data is changing constantly (usually through the insertion of new elements), the concept of a continuous data stream is more appropriate [21, 266]. Here, continuous queries present themselves as a natural interface to handle these data streams. Nonetheless, continuous queries are formalized for a wide range of environments [24].

More recently, some new database systems have appeared, offering some continuous queries features. For instance, InfluxDB[1] is a time-series database with support for continuous queries that run automatically and periodically on real-time data and store query results in a specified measurement (i.e., location). Contrasting with TARS, these continuous queries run periodically (with a defined frequency, e.g., every hour), and not as a response to changes in the data.

Another system, Apache Ignite[2] is a distributed in-memory database. Additionally, it is also a caching and processing platform designed to store and compute on large volumes of data across clusters of nodes. Here, the featured continuous queries monitor data modifications occurring in a cache, and notify users of all the data changes that fall into the query filter. However, its continuous queries send the whole updated object to the query issuer, which can lead to excessive network usage, especially if the object is very large. To overcome this limitation, users have to deploy special transformer predicates that are executed on the remote nodes for every updated object and send back only the results of the transformation.

In conclusion, TARS proposes fusing the storage substrate together with the P/S communication paradigm, thus providing persistent publications (i.e., data items) along with persistent subscriptions and also time-awareness. Characteristics that no previous works present simultaneously.

3.3 Building Synergies Between Storage and Publish/Subscribe

Typical storage systems provide a request/reply *proactive* interaction model for data retrieval, making it difficult to be aware of the available data, and requiring users to explicitly search for it. Also, in most P/S systems, publications are *transient*, i.e., once matched and disseminated, they are not further stored or processed. Thus, only subscribers online at the time of publication are notified.

To overcome such shortcomings, we build strong synergies between the storage substrate and the P/S paradigm. On the one hand, the storage substrate leverages the P/S abstraction to provide a *reactive* interaction model whereby users register their interests through subscriptions and are notified as new relevant data is generated, not requiring

[1] `https://www.influxdata.com/`
[2] `https://ignite.apache.org/`

them to be constantly searching for new data. On the other hand, the P/S abstraction takes advantage of the storage substrate to provide *persistent* publications, enabling the *time-awareness* concept and providing full time decoupling [85].

In the end, we devise the TARS data storage and dissemination model. Its storage interface provides the usual data store operations: insert, retrieve, and delete. Additionally, due to its integration with the P/S abstraction, it also offers the regular P/S operations: publish, subscribe, and unsubscribe. All operations are asynchronous, receiving results through callbacks (implemented as specific handlers).

3.3.1 Inserting Data

The typical insert operation allows data to be stored in the system. However, due to the integration with a P/S abstraction, here, the insert operation (of the storage substrate) is *fused* with the publish operation (of the P/S system). As a result, the insertion of a data object into storage may additionally trigger the sending of notifications to subscribers.

A *data object* is the basic unit of work and is seen as an opaque set of bytes. Every object has some associated *metadata* that consists in a tuple

$$\langle oid, T, s, ts^{\text{ins}}, nid \rangle$$

where:

- oid is the object identifier;
- T is a set of tags or keywords related with the object, e.g., hashtags used in social networks;
- s is a summary or a small description of the object, e.g., a thumbnail of an image or a video;
- ts^{ins} is the object insertion/publication timestamp; and
- nid is the owner's node identifier.

This metadata concept is also very flexible (and extensible). It basically can be seen as a small amount of data arranged as a set of key-value pairs. Thus, it can even be extended with application-specific information (i.e., other key-value pairs).

To avoid name collisions (among different nodes), the system-wide unique *object key* is the pair $\langle oid, nid \rangle$, composed of both the object and the owner's node identifiers.

Tags are used as topics for subscriptions, thus enabling a *topic-based* P/S system. Although topic-based addressing [187] is not as expressive as content-based systems [5], it requires far less filtering and computations, which fits our target environments populated by resource-constrained mobile devices. Nonetheless, this tagging feature provides a flexible annotation scheme. For instance, by adding the owner's node identifier to the tags of its own objects, an application can easily enable the retrieval of all the objects stored by a certain node/user.

3.3.2 Deleting Data

To support subscriptions within a time frame in the past, insertions (or publications) must be persistently stored within the system. Accordingly, this model also supplies an operation to enable data to be removed from storage.

Naturally, the delete operation removes an object from storage, thus making it inaccessible to future queries, i.e., subscriptions. However, note that likewise subscriptions targeting the past will not see deleted objects, even if these were initially available in the subscription's time frame.

Regarding a simple access control mechanism, only the owner of a data object can delete it (i.e., an object can only be deleted by the same node that inserted it in the system).

3.3.3 Querying Data

Since we make use of the P/S abstraction, in TARS querying data means subscribing to the desired tags. As a response, notifications will be received for data objects matching the issued subscriptions.

With time as a first order dimension, a subscription consists in a tuple

$$\langle sid, q, ts^{\mathrm{s}}, ts^{\mathrm{e}}, nid \rangle$$

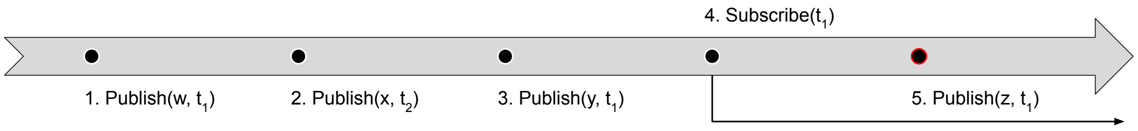
where:

- sid is the subscription identifier;
- q denotes the query that defines which tags are relevant;
- ts^{s} and ts^{e} are the timestamps defining when the subscription's time frame starts and expires, respectively; and
- nid is the subscriber's node identifier.

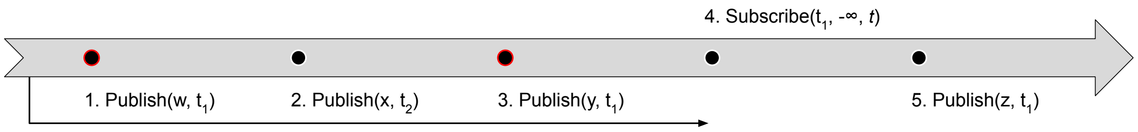
Unlike typical topic-based P/S systems, that only allow one topic per subscription, we support *arbitrary* propositional logic formulas where literals are tags associated with objects (e.g., '$A\,\&\,(B|C)$' captures objects tagged with A and at least one of B or C).

The ts^{s} and ts^{e} timestamps specify the subscription's time frame in which the subscription is active, where the special value $\perp$ represents, respectively, the times at which the system started and stopped to exist. Assuming a subscription is issued at time instant t, we have the following alternatives:

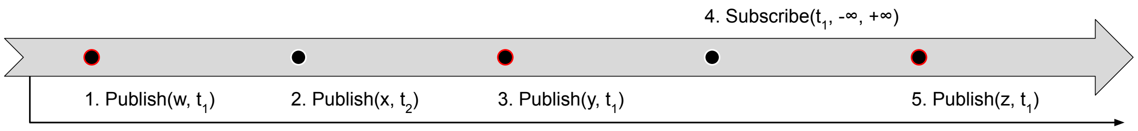
- $ts^{\mathrm{s}} = \perp \wedge ts^{\mathrm{e}} = t$ matches events that happened before the subscription (this allows a typical search or find operation on the data store, depicted in Figure 3.1b);
- $ts^{\mathrm{s}} = t \wedge ts^{\mathrm{e}} = \perp$ matches events after or concurrent with the subscription (like a standard subscription, depicted in Figure 3.1a); and



(a) Standard subscription.



(b) Standard query.



(c) Subscription in the past.

Figure 3.1: Comparison among a standard subscription, a standard query, and time-aware subscription in a generic topic-based publish/subscribe system. Publish(x, t) means publishing item x with topic t, and Subscribe(t, a, b) means subscribing to topic t between timestamps a and b. The red circles are publications that match the subscription.

- $ts^s = ts^e = \perp$ matches all the past and future events in the system (as depicted in Figure 3.1c, where we use $-\infty$ and $+\infty$ to represent $\perp$).

Notice that these parameters can also take any concrete timestamp value specifying a particular point in time. Additionally, depending on the specific system implementing TARS, these values can take various representations. For instance, they can be real clock timestamps (e.g., 2016-11-16 06:43:19.769), or just some (system-level) numeric representation of time (e.g., time instant 21).

As already mentioned, in TARS, a subscription has a specific time scope, i.e., every subscription has an assigned time frame defining when it should be active (and producing possible notifications resulting from matching publications during that time period). Figure 3.1 depicts the difference among a standard query, a typical subscription, and a time-aware subscription (as provided in TARS). A standard subscription, as in Figure 3.1a, only matches publications (the red circles) that are issued after the subscription is installed. Consequently, the subscriber is oblivious to such past publications, and has no way of accessing them. A standard query, as in Figure 3.1b, only matches publications that are issued before the subscription is installed. That is, the query only takes into account the data stored in the system before its execution. In turn, Figure 3.1c illustrates an example of a time-aware subscription that subscribes to all the past and future publications, thus also matching against all previous publications (here, we use the timestamps $-\infty$ and $+\infty$ to represent the beginning and the end of the system).

Due to the unreliable nature of our target (wireless) environments, subscribers are notified of all relevant data in a *best effort* manner. After a subscription, notifications may be triggered in two situations:

- upon an insertion, by detecting that the object being stored matches existing subscriptions; and
- upon issuing a subscription that spans into the past, by detecting that this new subscription matches previously stored objects.

Note that, to minimize the information passing through the network (and save bandwidth), notifications are sent to the respective subscribers carrying *only* the metadata of the matching objects (and not the entire data objects)—which is expected to be much smaller in size than the actual data object it concerns.

The (reverse) unsubscribe operation revokes a subscription before it naturally expires after its end timestamp, ts^{e}.

When issuing a subscription for a popular tag (i.e., with many indexed objects) that spans into the past, a subscriber might get flooded by a large amount of notifications (i.e., an excess of past notifications), which implies a large amount of communication. To attenuate this problem, when subscribing for a time frame in the past, a subscriber can specify a *notification policy* defining how many (and which) notifications it wants to receive. We define three possible policies:

1. **All** - the subscriber is notified about all the matching objects;
2. **Partial** - at most, the subscriber is notified about n of the most recent matching objects; and
3. **Random** - at most, the subscriber is notified about n matching objects chosen at random.

Additionally, for the *partial* notification policy, if interested, a subscriber then can request more of the matching objects' metadata, receiving the notifications in *explicitly requested batches* (similar to the concept of pagination). After, all subsequent matching objects will be notified as usual.

3.3.4 Retrieving Data

Through subscriptions, users are notified *only* about data they are interested in, allowing them to discover what data exist in the system in a reactive and asynchronous manner. Even so, a typical search/find operation can still be issued by subscribing to the desired query with timestamps $ts^{s} = \perp$ and $ts^{e} = now$ (§3.3.3).

Due to our reactive model, objects can *only* be retrieved as a response to notifications (using the received object metadata), thus revealing a close relation between the subscribe and retrieve operations. So, object metadata is the only information given to subscribers for them to decide if objects are relevant enough and proceed to retrieval.

Algorithm 3.1 TARS basic API.

```
procedure INSERT($obj, T, s, opHandler$)
    $obj$ - the object to be inserted
    $T$ - the set of tags related with the object
    $s$ - the object's summary
    $opHandler$ - the handler for processing the operation's outcome (i.e., the newly created data object)

procedure DELETE($oid, opHandler$)
    $oid$ - the identifier of the object to be deleted
    $opHandler$ - the handler for processing the operation's outcome (i.e., the specified data object)

procedure SUBSCRIBE($q, ts^s, ts^e, policy, opHandler, notHandler$)
    $q$ - the query (in propositional logic) defining the relevant tags
    $ts^s$ - the timestamp defining the start of this subscription's time frame
    $ts^e$ - the timestamp defining the end of this subscription's time frame
    $policy$ - the desired notification policy for this subscription
    $opHandler$ - the handler for processing the operation's outcome (i.e., the newly created subscription)
    $notHandler$ - the handler for processing notifications concerning this subscription

procedure UNSUBSCRIBE($sid, opHandler$)
    $sid$ - the identifier of the subscription to be removed
    $opHandler$ - the handler for processing the operation's outcome (i.e., the specified subscription)

procedure RETRIEVE($oid, opHandler$)
    $oid$ - the identifier of the object to be retrieved
    $opHandler$ - the handler for processing the operation's outcome (i.e., the requested data object)
```

Received notifications must be acted upon, and may either be discarded, trigger an immediate retrieve operation, or be stored by the application for later processing.

3.3.5 TARS API

As mentioned before, our time-aware reactive storage model provides five asynchronous operations: `insert`, `delete`, `subscribe`, `unsubscribe`, and `retrieve`. Algorithm 3.1 presents the TARS basic application programming interface (API) and we detail each operation signature and its parameters.

The `insert` operation allows data to be stored in the system. It receives four parameters: the data object to store, as an opaque byte sequence; a set of tags associated with the object; the object's summary, as an opaque byte sequence (usually much smaller than the object itself); and the corresponding operation handler, responsible for processing the operation's outcome. For each operation, its (operation) handler is parameterized with the type of the operation's outcome `T`, and has two procedures:

- `onSuccess(outcome: T)`; and
- `onFailure(reason: String)`.

For instance, in the case of the `insert` operation, its handler is parametrized with a `DataObject` type, that represents the stored object encapsulating both the given byte sequence and its associated metadata. In turn, for a `subscribe`, its operation handler is parametrized with a `Subscription` type, encapsulating all the information details of a subscription (e.g., query, time intervals).

Next, the `delete` operation allows the removal of a data object from the system. It receives only two parameters: the identifier of the object to be removed; and the operation handler (parametrized with the `DataObject` type).

As its name suggests, the `subscribe` operation allows the subscription for a set of tags. It receives a total of six parameters: the query, as a propositional logic formula, determining the relevant tags; the timestamps defining the subscription's time frame (start and end); the notification policy (which defaults to All); the operation handler (parametrized with the `Subscription` type); and the corresponding notification handler. The notification handler has only one procedure:

- `onNotification(subscription: Subscription, metadata: Metadata)`.

This is called when a new notification is received and has two parameters, the matching subscription and object metadata.

The `unsubscribe` operation is the inverse of the `subscribe` operation, and allows the removal of an ongoing subscription from the system. It receives two parameters: the identifier of the subscription to be removed; and the operation handler (parametrized with the `Subscription` type).

Lastly, the `retrieve` operation allows the retrieval of a data object given its identifier. It also receives two parameters: the identifier of the object to be retrieved; and the operation handler (parametrized with the `DataObject` type).

Note that, in this model, tags are *out-of-band* information, i.e., there is no way of knowing which tags are being used in the system at each time. Thus, clients have to gather that information from possibly another source. For instance, this can be an external source to the system, such as some independent storage service just for keeping track of tags. In turn, the system itself can also be used to store the tags being handled, e.g., through a reserved system tag—an approach followed by Ephesus and Basil (see §7.1 and §7.4, respectively).

3.4 Concluding Remarks

In this chapter, we present the concept of time-aware reactive storage, that fuses the P/S paradigm with the storage substrate, providing persistent publications and allowing queries (i.e., subscriptions) within a specific time scope.

Here, the insert operation of the storage substrate is merged with the publish operation of the P/S system, enabling applications to be notified as relevant data is generated and stored. Additionally, subscriptions allow arbitrary propositional logic formulas as queries, and have a time frame defining when they are active. The innovative characteristics of TARS offer a novel way for sharing and accessing data that has been previously stored, or is being generated in quasi-real-time.

3.4.1 Discussion

Overall, this TARS concept represents a fundamental *overhead shift*. Instead of requiring users to explicitly search for the available data, it allows users to register queries (with defined time boundaries), which are then notified as new relevant data is stored in the system—providing a reactive interaction model. As a consequence, the overhead from the stakeholders that actually benefit more from this approach—users requesting data—is reduced (compared with the explicit search approach), and is *moved* to the stakeholders that do not benefit directly from it—users that have the data and can provide it. Thus, as a peer-to-peer (P2P) solution, all the stakeholders share their resources among them. This can work as an argument against this approach. However, we reckon that users usually do not mind sharing their resources just for a greater good (e.g., P2P file sharing), specially if they can also benefit from what the systems have to offer.

Other compelling reasons are also the *volunteer computing* [15, 228] and the *crowdsourcing* [76, 116] hypes. Volunteer computing uses computing resources (e.g., processing power, storage, etc.) donated by the general public to do distributed scientific computing. Systems like BOINC [13] have been proved and tested throughout the years, being used by numerous scientific projects, e.g., SETI@home [14]. Results have also shown the general public massively adheres to this kind of initiatives and is willing to share their computing resources for a "greater good" [14]. Crowdsourcing is a type of participative online activity in which an entity proposes to a group of individuals the voluntary undertaking of a task entailing mutual benefit [83]. This idea has been extensively used as a cost-effective way of harnessing the collective power of multiple individuals. Inclusively, it even changed the way of working of various sectors of the world's economy [116].

With all these aspects in mind, it makes sense to crowdsource the computing and storage power of a collection of nearby mobile devices to support a new generation of applications. Furthermore, people have shown to be receptive to the idea of harnessing the individual resources in order to make sense of the old motto "unity is strength".

Additionally, as we will show in §7, this model and interface allows the easy development and implementation of a plethora of use cases.

3.4.2 Publications

The work reported in this chapter was presented in the following publications:

- **It's About Thyme: On the Design and Implementation of a Time-Aware Reactive Storage System for Pervasive Edge Computing Environments [241].** *João A. Silva*, Filipe Cerqueira, Hervé Paulino, João M. Lourenço, João Leitão, Nuno Prequiça. In Elsevier Future Generation Computer Systems (**FGCS**). 2021.

- **Time-Aware Reactive Storage in Wireless Edge Environments [245].** *João A. Silva*, Hervé Paulino, João M. Lourenço, João Leitão, and Nuno Preguiça. In Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems:

Computing, Networking and Services (**MobiQuitous**). Houston, Texas, USA, 2019.

- **Time-Aware Publish/Subscribe for Networks of Mobile Devices** [244]. *João A. Silva*, Hervé Paulino, João M. Lourenço, João Leitão, Nuno Preguiça. **arXiv**:1801.00297. 2017.

Chapter 4

It's About Thyme: TARS in Wireless Edge Environments

"Ever tried. Ever failed. No matter. Try again. Fail again. Fail better."
— Samuel Beckett

This chapter reports about data storage and dissemination for co-located mobile devices in settings without access to network infrastructures. More specifically, it reports about Thyme, a system implementing time-aware reactive storage (TARS) in wireless edge environments. In this work, we only address the lowest level of the network hierarchy, i.e., the end-user devices.

We start by giving some context and motivation in §4.1. Then, we review related work and compare our approach to different proposals in §4.2. Next, we give an overview of our system and its architecture in §4.3. After, we start detailing our two approaches, Thyme-LS in §4.4, and Thyme-DCS in §4.5. Then, we describe our Android implementation of Thyme-DCS, as a Java library, and the design of a photo sharing application developed on top in §4.6. Afterwards, we report a characterization of the scenarios best suited for the use of the proposed solutions, through a three-fold evaluation: via an analytical study in §4.7, and both simulation and real world experimentations in §4.8 and §4.9, respectively. We conclude this chapter with §4.10, by presenting our main findings, some future work, and the publications resulting from this work.

The work in §4.6 and §4.9 was developed in the context of the M.Sc. of Cerqueira [46], more specifically the implementation of the Thyme-DCS library for Android, the adaptation of the photo sharing application [243] to work on top of it, and the corresponding evaluation of the prototype.

4.1 Introduction

We are witnessing a rapid growth of both the capabilities and amount of mobile devices worldwide [56, 59]. As such, there is a wide adoption of smartphones and tablets for performing the most diverse activities, from leisure to work-related tasks. Hence, the volume of data generated by these devices, like user-generated content and sensor data, is also growing rapidly [57, 59].

Much of the data generated by mobile devices in all sorts of social gatherings (like sports events, protests, festivals, or ceremonies) has *localized* and *ephemeral* interest. People in such events are usually interested in similar types of information (e.g., statistics and videos at sports events), and such interest typically diminishes over time. Thus, swift and spontaneous data storage and dissemination among neighboring mobile devices can be of great usefulness. For instance, smartphones carried by people in such gatherings can collect lots of useful data that, when shared among co-located devices, may help others discover new points of interest, enjoy videos of special moments (from multiple viewpoints), or avoid waiting lines or crowded areas in a venue.

In many situations, making information available may be of paramount importance (e.g., disaster situations [166], military scenarios [70]), or just really helpful (e.g., crowded events [82]). So, being dependent on network infrastructure access to support such use cases may be unwise, or even unfeasible, due to their potential overload or destruction. Even assuming the availability of infrastructure, transferring large amounts of data to and from the cloud can lead to network congestion, various delays, and possible monetary costs. Furthermore, in those scenarios, mobile devices often experience poor or intermittent connectivity, leading to *availability* issues if application storage and logic are fully delegated to a remote cloud infrastructure. Still, the non-negligible costs associated with network infrastructure setup (e.g., adding access points) further motivates the need to have devices interact in a device-to-device (D2D) manner, through an infrastructure-less or ad-hoc network [286]. Thus, the main question we address in this chapter is: *how to support reliable and efficient data storage and dissemination among co-located mobile devices without resorting to centralized services and subsisting with no network infrastructure?*

The extensive proliferation of mobile devices at the edge of the network, along with the increasing growth of their capabilities, offers a massive computing and storage infrastructure of (still mostly) untaped resources. Together, the ubiquitous smart mobile devices, the opportunistic gathering of users, and the growing pervasiveness of edge computing environments [96, 240], have enabled novel opportunities for data storage and dissemination at the *network edge*. In fact, in this kind of localized environments, it is more efficient to communicate and distribute information among *nearby* devices than to use distant centralized intermediaries [110, 117]. By storing data near its source (e.g., where it is generated), applications can be more responsive while *relieving* some of the load from cloud and network infrastructures, potentially also providing increased data privacy and ownership.

Allowing systems' components to actively and directly collaborate at the edge requires some form of distributed data repository as to share and disseminate information. Thus, we propose Thyme, a novel TARS system (§3) for networks of mobile devices, that exploits *synergies* between the storage substrate and the publish/subscribe (P/S) communication paradigm. As such, queries are in the form of *subscriptions* that have a specific *time scope* defining when they are active (and can even include the past). Leveraging this novel time-aware abstraction, Thyme is able to achieve robust, efficient and timely data storage, dissemination, and querying. It also allows both the notification and retrieval of desired data with low overhead and latency, using limited bandwidth and while being resilient to possible message losses and node failures.

Implementing a system providing the TARS interface can be achieved through a multitude of ways. In this work, we present two different approaches to Thyme:

1. Thyme-LS is the simplest lightweight implementation we could envision, serving as a baseline. It follows a lightweight, yet effective, unstructured approach using local storage and query flooding; and

2. Thyme-DCS is more intricate, inspired by specificities of wireless environments. Namely, it is inspired by the fact that geographical positions have a close relation to topology in wireless networks, and follows a data-centric storage (DCS) approach [212], whereby we build a storage substrate over a geographic hash table (GHT) [17].

We implement both approaches in the ns-3 network simulator [216]. Moreover, we also address the application of Thyme to networks of real mobile devices, implementing Thyme-DCS as a library for Android devices, and developing a proof-of-concept photo sharing application on top.

Although some previous systems present in the literature offer some features similar to Thyme (e.g., tuple spaces [165, 202] or peer-to-peer (P2P) systems [79, 249]), none provides the same overall characteristics for this kind of environments (as we detail in §4.2, comparing Thyme against other proposals). Thus, to the best of our knowledge, Thyme is the first system to provide reliable reactive storage for pervasive edge computing environments that may be effectively and efficiently used in either small, medium and large scale scenarios.

In summary, the main contributions of the work presented in this chapter are the following: 1) the design of Thyme, a novel time-aware reactive storage system, and our two approaches to this proposal—the unreliable Thyme-LS, and the reliable Thyme-DCS; 2) the implementation of the Thyme-DCS approach, as a Java library, for sharing and storing data in networks of Android mobile devices (implemented in the context of the M.Sc. of Cerqueira [46]); and 3) the characterization of the scenarios best suited for the use of the proposed solutions, through a three-fold evaluation: via an analytical study, and both simulation and real world experimentations.

Table 4.1: Comparison of Thyme with publish/subscribe systems.

	Environment	Time Assignment	Substrate
[55]	Wired	Subscriptions	REBECA [183]
[277]	Wired	—	DB
GeoRendezvous [41]	Wireless	—	GHT [17]
Chapar [137]	Wireless	Publications	OLSR [60]
Thyme	Wireless	Pub. & Sub.	GHT [17]/Flooding

4.2 Related Work

Concerning work related to our proposal, we address and compare against three main categories: P/S systems, data storage and dissemination systems in general, and the particular case of tuple spaces.

4.2.1 Publish/Subscribe

As already mentioned, the notion of publication persistence has not been addressed in most systems. However, some approaches for wired settings exploit the concept of a persistent data repository, by means of distributed buffers [55] (allowing only to specify how many data items to request from the past when subscribing) or by integration with traditional databases [277] (without any notion of time).

Also, there have been many proposals of P/S systems for wired settings, such as Kafka [141] from LinkedIn, clearly demonstrating the feasibility and potential of this communication paradigm. Nonetheless, such solutions do not consider time as a first order dimension of the P/S abstraction. Furthermore, solutions for wired scenarios cannot be easily adapted for wireless setting where connectivity is not stable.

Table 4.1 highlights the main aspects when comparing our proposal with other P/S systems. GeoRendezvous [41] is a P/S system for wireless networks that uses the same GHT as Thyme as a substrate. However, it does not provide time-awareness nor publication persistence. Additionally, their use of the GHT is fundamentally different. This approach combines the GHT with multiple hash functions to hash topics to different cells and allow subscribers to choose the closest cells to themselves, thus possibly reducing latency. This is very similar to the way Thyme uses the GHT to retrieve objects from the closest replicas to the issuer. In turn, contrary to Thyme, it only allows one tag per subscription. It also requires the periodic advertisement of subscriptions and publications to react to changes in the topology.

In the case of wireless settings, to the best of our knowledge, Chapar [137] is the only P/S system that addresses persistent publications. However, it only assigns time to publications, which are buffered only until their lifetime expires. In Thyme, subscriptions have their time scope assigned. In turn, publications are permanently stored, and they may be deleted from the system upon request. Thus, new subscribers can always request

Table 4.2: Comparison of Thyme with data storage and dissemination systems.

	Infra.	Data Avail.	Symmetric	Substrate
Krowd [79]	Yes	No	Yes	1-hop DHT
Ephesus [243]	Yes	Yes	Yes	DHT
MobiTribe [268]	Yes	Yes	No	Central Server
PAN [162]	No	Yes	No	Prob. Quorums
Phoenix [195]	No	Yes	Yes	Simple Quorums
iTrust [157]	No	No	Yes	Random Walks
PDS [249]	No	±	Yes	ICN [285]
Thyme	No	Yes	Yes	GHT [17]/Flooding

previously published data. Also, Chapar is not functionally symmetric, demanding more work from broker nodes, and thus achieving poor load balancing, an aspect that has been explicitly considered in the design of Thyme.

4.2.2 Data Storage and Dissemination

Table 4.2 highlights the main aspects when comparing our proposal with other general data storage and dissemination systems targeting various environments. Krowd [79] and Ephesus [243] enable content sharing and storage among nearby mobile devices. While Krowd relies on a one-hop distributed hash table (DHT), requiring each device to know every other device in the network, Ephesus is sustained by a classical DHT (requiring only partial knowledge of the network). As an handicap, they both require some kind of network infrastructure for inter-device communication (e.g., an access point). Of the two, Ephesus is the only one to address device mobility or failure, and data availability, via replication. In turn, Thyme supports several wireless technologies, and targets multi-hop environments using a GHT, known for being more suitable in wireless networks. It also employs several replication mechanisms to address mobility and data availability.

MobiTribe [268] is a system for content sharing on mobile devices, across the Internet. It uses a central server for content discovery, peer registration and metadata management. Data is replicated in several peers according to their interests, and it uses prefetching techniques to improve retrieval latency.

PAN [162] and Phoenix [195] are two systems for reliable data storage in mobile ad-hoc networks (MANETs). PAN is an asymmetric system based on probabilistic quorums, where peers have specific roles (thus, some may perform more work than others). In turn, Phoenix uses a round-based simple quorum protocol for one-hop networks only.

iTrust [157] and PDS [249] focus on data discovery and retrieval on co-located devices. iTrust is based on random walk techniques for propagating both metadata and queries through the network. As Krowd, it also does not address data availability. In turn, PDS is inspired by information-centric networking (ICN), and targets small networks with low to moderate mobility. Its aggressive caching policy can lead to serious storage overheads,

Table 4.3: Comparison of Thyme with tuple spaces systems.

	Purpose	Data Sharing
TuCSoN [188]	Internet	Constant
LIME [202]	Federated TS	Transient
TOTA [165]	Autonomous Propagation	Rule-based
Thyme	Wireless	Constant

and since data is only cached if requested, less popular data may even disappear.

All these systems employ the *request/reply* interaction model, whereby peers have to proactively search for content. In turn, Thyme implements TARS and explores synergies between the P/S paradigm and the storage substrate, to provide both persistent publications and a reactive interaction model, thus allowing applications to react to new data being generated and stored. At the same time, it presents the potential to decrease network traffic, as users do not need to be constantly searching for the desired data.

Other approaches based on opportunistic and delay-tolerant networking [73, 194, 259] provide communication and content sharing in the presence of intermittent connectivity and other harsh conditions. They take advantage of opportunistic contacts between peers to allow the exchange and spread of information. This means content dissemination is best effort, i.e., information spreading depends on the availability and willingness of interested peers to carry such content. Some of these systems also provide a reactive interaction model for data retrieval. They are, however, devised for extreme environments that relax temporal restrictions to the order of hours or days, something not feasible for the kind of use cases we target.

4.2.3 Tuple Spaces

Table 4.3 highlights the main aspects when comparing our proposal with other tuple spaces systems. TuCSoN [188] was designed for mobility in Internet environments and presents the notion of programmable tuple spaces (spread over Internet nodes). Tuple spaces are enhanced in that their behavior in response to agent's operations can be extended so as to embody application-specific computations. These tuples spaces are rather complex and cumbersome to reason with. Furthermore, it is not easily adaptable to dynamic wireless environments (e.g., it assumes reliable communication), and its main focus in on the programmability of the (coordination) tuple space.

LIME [202] breaks up the notion of a global tuple space, and distributes its content across multiple mobile components. When components are within range (i.e., mobile agents are on the same host or communication is available between mobile hosts), the contents of the tuple spaces held by the individual mobile components are transiently shared, forming a federated tuple space. The contents of these virtual tuple spaces evolve in time according to the current connectivity pattern. Although reactions enable tuple spaces to react to the insertion of relevant tuples, they are sensitive to hosts' connectivity, since the federated tuple space only takes into account data spaces from components within range. In several mobile and wireless environments, this approach is not sufficient to generally support distributed services or applications. Therefore, LIME was devised for small scale scenarios. In turn, Thyme leverages a lightweight flooding approach or a GHT for ensuring the best possible connectivity in large-scale scenarios, and its routing schemes jointly with its replication mechanisms allow the matching of publications against subscriptions of all peers in the network.

In TOTA [165], tuples are injected and can autonomously propagate into the network according to specific rules (i.e., they are not assigned to specific tuple spaces). Each tuple is formed by: its content data; a propagation rule, the policy by which the tuple is cloned and diffused across the network and how its content should change during propagation; and a maintenance rule, the policy whereby the tuple content should change due to events or time elapsing. Propagation consists in a tuple cloning itself, being stored in the local tuple space, and moving to neighbor nodes recursively. Tuples are not necessarily distributed replicas. According to their propagation and maintenance rules, they can assume different values in different nodes (expressing some kind of contextual or spatial information). In the end, unlike traditional event-based models, tuples propagation is not driven by a P/S schema, but is encoded in the tuples' propagation rule. By constantly monitoring the network local topology and the insertion of new tuples, TOTA can automatically re-propagate or modify the content of tuples as necessary conditions occur. Subscriptions only react to changes in a node's local tuple space (or from its one-hop neighbors). To achieve something similar to Thyme, data should be propagated to every network node in order for subscriptions to be matched against the data. Otherwise, some nodes would not be notified about relevant data. TOTA also requires every node to execute the matching of subscriptions against tuples, thus suffering from redundant work and poor load balancing. In contrast, Thyme allows for better load balancing, distributing the load when matching subscriptions and publications.

4.2.4 Others

Regarding *app stores*, there are several applications for sharing files between mobile devices. For instance, SuperBeam [156] and Xender [16] allow synchronous one-to-many data exchange. However, data is only available while its owner is online.

There are still other applications [38, 190] and specialized devices [102] that provide ad-hoc (multi-hop) communication among mobile devices, allowing data dissemination when network infrastructures are inaccessible.

4.3 The Many Leaves of Thyme

The design of a system implementing TARS for wireless edge environments presents a set of interesting challenges. For example, where to place data and how to find it? What are the proper trade-offs between communication and reliability? How and what data to disseminate (and when)? In the end, Thyme's design, that we present next, considers these and other issues.

4.3.1 Use Cases

Thyme can be used to build generic data dissemination services for the kind of environments we are targeting. We argue that Thyme fits perfectly in scenarios where big crowds are gathered, using their mobile devices to collect data (e.g., photos, video, text) and share that same data with people in their vicinity, akin to social networks [237].

Consider, for instance, a scenario where spectators in different parts of a football stadium may share their views of the game through self-generated multimedia content. In this case, spectators would be able to see key moments of the game from multiple viewpoints, including those of the spectators in key locations or closer to the field. Offering such possibility can significantly improve user experience—something football teams are willing to invest in, if it means they will attract more fans to their stadiums [289].

In fact, this kind of augmented user experience is already being explored by some companies [289], using the venues' existing fixed communication infrastructures, which are single points of failure that may be subjected to overload conditions and other failures (e.g., power outages [82]). In turn, the pervasiveness of mobile devices and the advances in the edge computing paradigm offer the possibility to provide such enriched user experience with negligible cost for infrastructure managers, while at the same time, working to alleviate the load on those infrastructures.

4.3.2 System Model

We consider a classical asynchronous model comprised of $\Pi = \{n_1, \ldots, n_k\}$ mobile devices (hereafter called nodes) with no mobility restrictions, other than those imposed by the venue they are in and the natural speed limits of humans[1]. Our algorithms do not assume any radio technology or routing infrastructure, being practical in several wireless settings. Nodes communicate by exchanging messages through a wireless medium (e.g., Bluetooth, Wi-Fi ad-hoc, Wi-Fi Direct, Wi-Fi Aware) [217, 242, 264, 265], and have no access to any form of shared memory. Nonetheless, nodes should be able to establish (point-to-point) communication channels with (all) their one-hop neighbors, and thus need to have some kind of discovery mechanism in order to determine their neighbors. We also consider the classical crash-stop failure model, whereby nodes can fail by crashing but do not behave maliciously.

In this (first) approach, data objects are considered immutable (however, currently, this is no longer the case; see §7.5). Also, we do not consider security or access control concerns, thus only publicly shareable data is manipulated (e.g., as in social networks).

[1]The record for top speed achieved by a human is 12.4 m/s, by Usain Bolt, seen during the 100 meters final of the 2009 World Championships in Athletics, in Berlin.

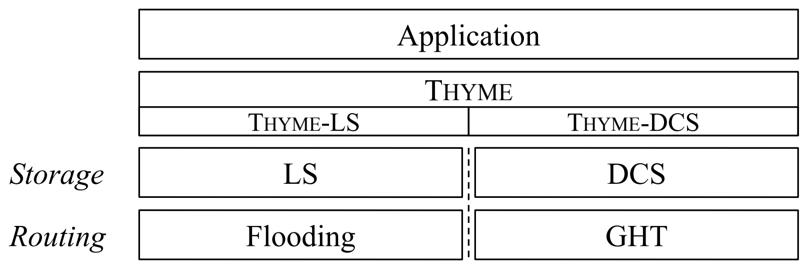


Figure 4.1: Thyme layered system overview.

Due to the unreliable nature of wireless communication mediums, Thyme notifies subscribers of all relevant data as completely and faithfully as possible, i.e., missing some notifications is permitted because applications are not expected to be mission-critical.

Each node has a globally unique identifier and can determine its geographical position, through GPS or other means [206]. Thus, nodes can be aware if they are moving or not. We also assume nodes' clocks to be synchronized (with a negligible skew). Both these assumptions are reasonable since we target mobile devices (e.g., smartphones) and nowadays even low-end devices come equipped with GPS and synchronize their clocks with the network providers, while other solutions allow device location even indoors [206].

4.3.3 Architecture

Since we target decentralized networks based on battery-constrained devices, load balance is a top-level concern. As such, in Thyme, akin to (flat) P2P systems, nodes are functionally symmetric and share the same responsibilities. This means that there are no centralized or specialized components (like P2P super-peers or P/S brokers), and each node can be a publisher, a subscriber, or both.

Thyme's design comprises three main layers, depicted in Figure 4.1. The bottom layer handles message routing. The middle layer is the storage substrate. The top layer is Thyme itself, providing its interface for applications.

As illustrated in Figure 4.1, we propose two different approaches for the two bottom layers (namely, routing and storage). Thyme-LS (§4.4) uses nodes' local storage, and query flooding, thus data objects are stored locally by their owners, while subscriptions are fully replicated in every node of the system. Its routing layer provides flooding to the entire network (using UDP broadcast), and (multi-hop) unicast using a typical ad-hoc routing protocol (e.g., DSDV [200], OLSR [60]).

In turn, Thyme-DCS (§4.5) follows a DCS approach [212], using a simple key-value storage substrate that we built over a cell-based GHT for wireless networks [17]. Its routing layer is materialized by this GHT, called cell hash routing (CHR). In a GHT, nodes self-organize according to their geographical positions and keys are hashed to physical locations. The node responsible for a key is the one closest to the key's geographical position. As represented in Figure 4.2, the physical/geographical space where the system is to be available is divided into a grid, i.e., into equally-sized square-shaped cells, and

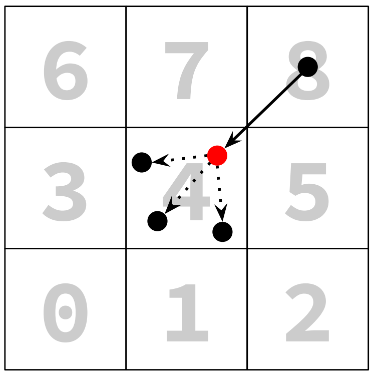


Figure 4.2: A geographic hash table and its virtual nodes.

all physical nodes within a cell collaboratively act as a virtual node. Since nodes need to map any point in space into its corresponding cell, cell size and some origin of space must be agreed beforehand. However, it suffices that every physical node is able to reach every other node in its cell and at least another node in adjacent populated cells to ensure the correctness of this solution. Messages are addressed to geographic locations, thus routed to the cell that contains the message destination. In turn, messages addressed to a cell are delivered to all physical nodes within the cell (similar to [150]). For instance, in Figure 4.2, a message addressed to a location inside the boundaries of cell 4 is received by the red node (chosen randomly by the routing protocol of the GHT; see §4.5.6.1), and then is forwarded (or broadcasted) to all the other neighbor nodes inside the cell. The use of the GHT is two-fold: 1) cells are used to store or index all the system data (data objects, its metadata, and subscriptions); and 2) cells are exploited to match subscriptions and objects, i.e., cells act as virtual P/S brokers.

Wireless communication mediums are known to be subject to many forms of interference, hence messages may be lost and not reach their final destination. However, as a design principle, this layer does not provide any mechanisms to recover from lost messages on the wireless medium, delegating this responsibility to the upper layers (abiding by the end-to-end argument [222]).

In both approaches, we used a plain in-memory hash table as the native object store (adapted to the implementation programming language). However, any other storage engine could be used (e.g., (in-memory) databases).

4.4 An Unstructured Approach: Thyme-LS

Thyme-LS employs a lightweight unstructured approach and has no extra maintenance overhead. It uses nodes' local storage, and query (in our case, subscription) flooding.

Both insert and delete operations are entirely executed locally. Thus, objects are only stored by their owners. On the other hand, Thyme-LS uses *subscription flooding* as its event routing strategy (like Gryphon [5, 23] and SIENA [44]). Hence, subscribe and unsubscribe operations are flooded and executed in every node, so subscriptions are fully replicated across all the system. These operations are broadcasted to all its one-hop neighbors who, then, forward the message to all their one-hop neighbors and so forth. Nodes keep track of received messages so that the ones already forwarded will not be sent again. Since every node has the complete set of all the system-wide subscriptions, the matching between objects and subscriptions is completely local.

In this approach, notifications may be triggered in two occasions:

- upon an insert operation, if that new object matches any of the node's locally stored subscriptions; and
- upon issuing a subscription (when flooding the respective message), each node that receives a subscription checks if it matches any of its locally stored objects.

Retrieve operations request the desired objects directly from their owners, using the information received in the notifications, and the multi-hop unicast communication primitive provided by the routing layer (§4.3.3).

Here, node mobility is handled in a completely transparent way by the underlying protocol used by the routing layer. Thus, nodes can move freely and the routing protocol takes care of all the necessary changes that come from that movement in order to continue to provide (the best possible) connectivity. Also, since data objects are only stored locally by their owners, Thyme-LS does not guarantee objects' persistence once their owners fail or leave the system.

When joining the system, nodes broadcast a join request. To avoid a flooding of replies, only a few neighbors (randomly selected through a coin toss) respond back with their locally stored subscriptions. Also, to minimize collisions, replies are delayed a (configurable) random amount of time. If no replies are received after a maximum number of retries, the joining node assumes it is alone, and starts operating as normal.

4.5 A Structured Approach: Thyme-DCS

This approach leverages heavily on the notion of cell (or virtual node) conveyed by the GHT used in its routing layer (§4.3.3). By using geographical information, Thyme-DCS boasts two complementary aspects: 1) it provides topology-awareness by design; and 2) allows the inference of the location of relevant data to subscriptions, enabling access to such data using a location-aware strategy.

4.5.1 Inserting Data

When executing an insert operation, this approach leverages on the cells conveyed by the underlying GHT. Object data and metadata are managed differently. The latter is indexed (and, thus, replicated) in all the cells resultant from hashing the object tags. In turn, the actual object is replicated in all the nodes of the owner's cell (see §4.5.2). This

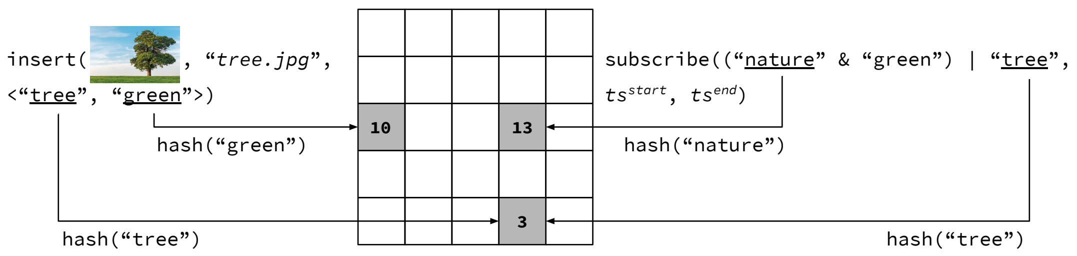

insert(, "tree.jpg", <"tree", "green">)
hash("green")
hash("tree")
10
13
3
subscribe(("nature" & "green") | "tree", ts^start, ts^end)
hash("nature")
hash("tree")

Figure 4.3: Insert and subscribe operations in Thyme-DCS. The tags' hashing determines the cells responsible for managing the object metadata (cells 3 and 10) and the subscription (cells 3 and 13). If a subscription has matching tags with an object, it will also have overlapping (responsible) cells, guaranteeing the matching and sending of notifications to the subscriber.

ensures only a small amount of data (i.e., the metadata) is sent through the network, whereas the bulk of the data is kept near its source.

Figure 4.3 illustrates an insert operation of a photo with identifier "tree.jpg", and tags "tree" and "green". The cells resultant from hashing each tag are responsible for managing the object's metadata and checking if subscriptions match this object.

4.5.2 Replication

Since we target highly dynamic environments, to provide data availability and churn tolerance, this approach employs replication mechanisms: active and passive replication.

4.5.2.1 Active Replication

Active replication takes advantage of the virtual nodes provided by the cell-based GHT. Upon an insertion, an object is disseminated inside the owner's cell. Onward, every node inside the cell should be able to reply to retrieve operations for that object. This ensures tolerance to churn and guarantees that stored content will remain in the system even if their owners' leave. Note that the objects' metadata is also (actively) replicated in the cells resultant from hashing the objects' tags (§4.5.1).

4.5.2.2 Passive Replication

In turn, similarly to seeders in P2P file sharing, passive replication leverages on the nodes that already retrieved an object to provide more replicas scattered in the network, increasing data availability. Thus, the number of locations from where an object can be retrieved grows in direct proportion to its popularity, and the very act of retrieving an object makes it more widely available.

4.5.2.3 Replication List

To enable both mechanisms, the system needs to keep track of the whereabouts of each object replica. This is achieved by listing an object's replica locations in its metadata, in what we call *replication lists*, L_{rep} (a list of pairs with node identifier and cell address—$\langle nid, cid \rangle$). Thus, in this case, the object metadata consists of a tuple

$$\langle oid, T, s, ts^{\text{pub}}, nid, \boldsymbol{L_{\text{rep}}} \rangle$$

These replication lists are bound to a (configurable) maximum number of locations, maintaining only the most recent entries. Also, a list only contains one entry for an object's active replica, representing all the nodes inside that cell (and this is a permanent entry on the list despite its recency).

Since nodes can *move*, their location may change over time. Hence, after a node stabilizes in a (new) cell, it must update its location for the passive replicas of the objects it holds (through some kind of location update messages). This update is only carried out for the cells managing the object metadata, i.e., the metadata of individual passive replicas is not actively updated.

4.5.3 Deleting Data

In the delete operation, the object metadata indexed by the object tags is removed from the responsible cells. However, while active replicas are also explicitly removed, the same does not happen to passive ones. Nonetheless, since the object metadata is removed (and with it, so is the replication list), the passive replicas become inaccessible and thus stop working as such.

4.5.4 Querying Data

Since the GHT used by Thyme-DCS only routes messages to geographical positions, there is the need to know where to send notifications, i.e., the node's address is not enough. Thus, subscriptions are extended with the location (i.e., the cell address) of the subscriber node, cid. Naturally, this information needs to be updated every time the subscriber node changes its cell. In the end, a subscription in Thyme-DCS consists of a tuple of the form

$$\langle sid, q, ts^{\text{s}}, ts^{\text{e}}, nid, \boldsymbol{cid} \rangle$$

4.5.4.1 Divide and Conquer

Leveraging on the fact that every propositional logic formula has an equivalent one in disjunctive normal form (DNF), we employ a simple divide and conquer strategy of breaking the disjunction into its individual conjunctive clauses, and evaluate each one separately. For a match to occur, it suffices that one evaluates to true. The use of DNF enables load balancing when matching objects against subscriptions, since the work can be split among different cells (and consequently nodes), each evaluating only one of the query's conjunctions. Additionally, it minimizes the amount of information transmitted to the responsible cells (by sending only a subset of the query, i.e., the relevant conjunction, to the responsible nodes).

For each conjunction, we randomly select as its key one of its *positive* literals (what we call *conjunction keys*). Hashing that literal determines the cell where to send that part of the query. That cell becomes a (virtual) broker for the subscription, and the nodes in the cell are responsible for checking if objects match the subscription, and notifying the subscribers, if need be. Thus, this approach employs the *rendezvous-based event routing* approach (like Scribe [45] and Hermes [203]). Figure 4.3 depicts a subscription of a query with two conjunctions. For each, one of its literals is chosen as its key, and determine which cells will become the virtual brokers for the subscription (in this case the two conjunction keys are "nature" and "tree").

For instance, assume the following query, already in DNF:

$$(A\&B\&E) \mid (A\&\neg C) \mid (D)$$

The disjunction is divided into its three conjunctions: 1) $A\&B\&E$; 2) $A\&\neg C$; and 3) D. Due to the restrictions already mentioned, conjunctions 2 and 3 have their keys automatically determined (literals A and D, respectively). But, any literal in conjunction 1 may be chosen to be its key (thus, we opt to choose one at random).

Even that, in some cases, the conversion to DNF can lead to an exponential explosion of the formula [71], we argue that most ordinary users do not make use of complex queries nor logic operators that regularly. Thus, we do not expect this to be an issue in practice and the possible conversion to DNF should be efficient.

4.5.4.2 Notifications

After a subscription, there are two occasions that may trigger notifications:

- upon an insertion, cells indexing the object metadata by its tags are responsible for checking if the new object matches any existing subscriptions stored locally; and
- upon a subscription, cells indexing the subscription by its conjunction keys are responsible for checking if the locally stored metadata match that new subscription.

When we break the subscription query into its multiple conjunctions, it suffices that one of the conjunctions evaluates to true, for a match to occur. However, since the conjunctions are evaluated by (probably) different cells, when different conjunctions of a same subscription both evaluate to true, the subscriber will receive duplicate notifications for the same matched object.

Lets assume the query given before, $(A\&B\&E) \mid (A\&\neg C) \mid (D)$. When a match with object $\mathcal{X}$, with tags A and D, is verified, both conjunctions 2 and 3 are evaluated to true. Consequently, two notifications will be sent to the subscriber (one from each cell that verified the match). We embrace this byproduct of our divide and conquer approach in two ways. First, we treat these duplicates as a positive outcome, because this (small) redundancy provides, to some degree, tolerance to lost messages. Second, we employ a duplicate detection in the subscriber side (and drop duplicate notifications).

4.5.4.3 Moving Subscribers

When a subscriber moves to a different cell, it must update its location for every active subscription it owns. During this situation, notifications sent to moving subscribers may never reach their destination. In such cases, the underlying routing layer returns negative acknowledgements (NACKs) for messages addressed to *individual* nodes that could not be delivered (see §4.5.6.4). NACKs are used to convey that a node is no longer in its supposed cell, which may be caused by movement or node failure.

Node movement will be detected through the subscriber's location update[2]. In such case, Thyme-DCS can re-send the notifications that were not previously delivered. Otherwise, after a (configurable) maximum waiting time, Thyme-DCS assumes the node has failed, and simply stops sending notifications. In case the node did not fail, and was just a straggler, it will have to re-issue all its subscriptions.

4.5.4.4 Unsubscribing

When executing this operation, unsubscribe messages are sent to the cells determined by hashing each conjunction key of this subscription, and the specified subscription is removed from storage.

4.5.5 Retrieving Data

Retrieve operations leverage the replication mechanisms in order to optimize from where to request an object. From all the locations in the replication list (§4.5.2.3) received in the object metadata (with the notification), the requesting node chooses the geographically closer replica, and sends a retrieve request for the desired object, as Figure 4.4 illustrates. If a negative reply is received, the requester proceeds and tries to retrieve the object from the next closest location in the replication list, until no more options are available, or a maximum number of retries is reached. As a last attempt, the cell *actively* replicating the desired object will be used (if not already tried), because it offers higher chances of success compared to every other replica.

One interesting aspect of using geographical routing is that it becomes easier for nodes to make hints on which replicas are better (i.e., closer), using the geographic distance as a metric. Since geographical positions have a close relation to topology in wireless networks, we expect this approach to minimize the distance data has to travel in the network, allowing for this location-aware strategy when retrieving objects.

4.5.6 Storage Substrate & Routing Layer

The major drawbacks of a routing protocol based on a DHT for wireless networks are the mismatch between the logical and physical topologies, and the high maintenance

[2]In fact, these location updates can be merged with the updates for the passive replicas (§4.5.2.3), reducing the amount of communication needed and the occupancy of the wireless medium.

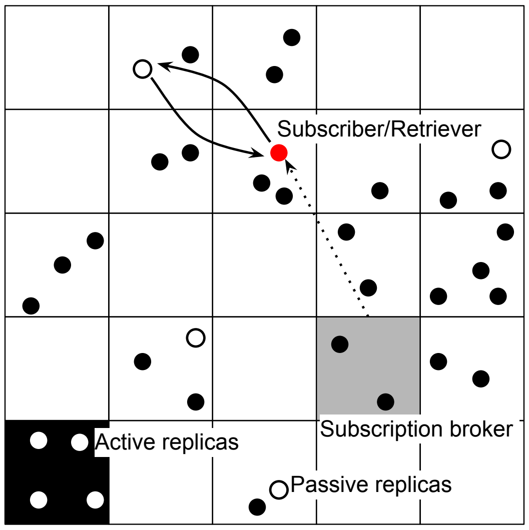


Figure 4.4: Notification and retrieve operation in Thyme-DCS. The dotted arrow is a notification sent to the subscriber. The other arrows represent a retrieve operation (request and reply from the closest replica).

overheads [291]. Inspired from both wired [150, 192] and wireless [17, 236] settings, we adopt a cell-based GHT as our routing protocol: CHR. By using geographic information, there is no mismatch between the logical and physical topologies [291]. Also, by leveraging on the control traffic of the underlying geographic routing protocol, the GHT does not add any other maintenance costs. At the same time, the cell-based approach relaxes the requirements for location accuracy (i.e., nodes only need to know their cell and to reach at least one neighbor in each of the populated adjacent cells), and is more robust to topology changes (requiring no action as long as nodes move inside their current cells).

In their essence, DHTs only provide routing. Hence, we implement a DCS substrate on top of this GHT, providing a simple key-value storage abstraction. Data items are named, and both their insertion and retrieval are performed using those names. To make this layer more suitable for the highly dynamic environments we target, we introduce several mechanisms and optimizations.

4.5.6.1 Routing

Our routing scheme is very similar to the ones used in [17, 212]. Routing is done at cell-level, using a variation of the greedy perimeter stateless routing (GPSR) protocol [135]. GPSR makes greedy decisions, forwarding messages to the next neighbor geographically closer to the message destination (thus, called greedy forwarding). When such strategy is not possible, the algorithm resorts to a recovery mode (called perimeter mode) that forwards messages around the voids in the network.

Lets assume node S in cell x wants to send a message to location D in cell y. If S wants to forward the message to a neighboring cell, it picks a random node from the cell and sends the message to that node. When in greedy mode, a cell selects as the next hop, another populated adjacent cell such that it minimizes the distance to the destination. However, if a cell is a local minimum on the path between source and destination, the cell must send the message in perimeter mode to go around the face in the direction where the line between the cell and destination lies. As soon as the message reaches another cell where the distance between it and the destination is less than that of the previous local minimum, the message leaves perimeter mode and reenters greedy mode. If a message in perimeter mode passes through the same cell h twice (without having switched to greedy mode in between), this means that the destination cell is empty or unreachable, and is inside the face contoured by the message (called the *home perimeter*). Therefore, cell h is the proxy cell of the destination, also called the *home cell*.

Here, cell size must be chosen in a way that maximizes the chance nodes inside a given cell can listen to all the nodes in any of the eight neighboring cells. Also, cell size cannot be too small or no gain will result from clustering.

For forwarding messages from cell to cell, we use unicast in order to take advantage of the (per hop) MAC-level retransmission mechanism. This layer provides: 1) a routing mechanism between cells; 2) routing to an individual node (in a specific cell); and 3) to broadcast messages within the context of a single cell[3]. In our implementation, the one-hop broadcast primitive is used as the neighbor discovery mechanism—transmitting *periodic beacons* with the node's current cell—, and also as the intra-cell communication primitive. Since broadcast is not acknowledged at MAC-level, this makes it a best effort communication primitive.

4.5.6.2 Dynamic Cell Structure

It is impossible to ensure that every cell is populated. Thus, we address empty cells forcing keys to take an entire loop around those cells [17, 212], stopping in the cell closest to the supposed destination (which becomes a *proxy* of the key's destination cell). This raises another problem when nodes populate previously empty cells, or leave the system and make some cell empty. So, a cell becoming empty has to deliver all its keys to its proxy cell. In turn, a cell becoming populated needs to receive its keys from its proxy cell, and also all the keys of the empty cells for which it now becomes the new proxy.

If two geographically independent clusters of cells connect at some point in time, GPSR and this proxy logic will trigger a rearrangement of the cells structure and its data. Eventually, the two clusters will merge and cell data will stabilize in its due location [212].

4.5.6.3 Mobility Awareness

We argue that moving nodes render routing information too *volatile*. Thus, in sharp contrast with CHR and GPSR, our routing layer is mobility-aware, i.e., only *stationary* nodes actively participate in message routing.

[3]Although the broadcast is received by other nodes in range, the message is filtered out at the routing layer.

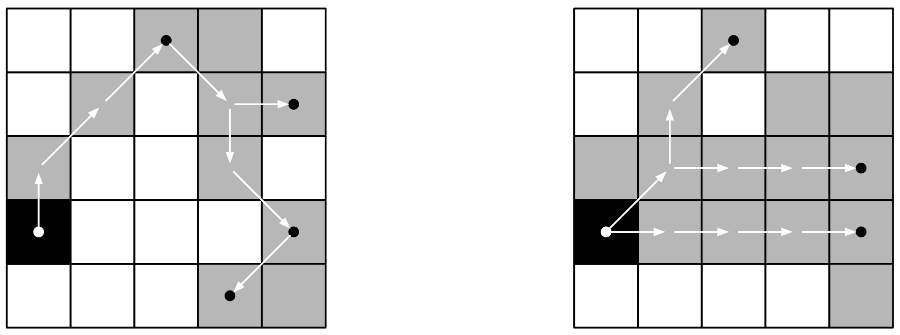

(a) Sparsely populated scenario. (b) Densely populated scenario.

Figure 4.5: Message destination aggregation working examples. The dark squares are populated cells, and black dots are the multiple message destinations.

Since our target scenarios have mild mobility patterns (i.e., nodes do not move constantly, and some might not even move during the entire event), *only* stationary nodes form the GHT. When a node starts to move and leaves its current cell, it stops participating in the routing protocol (i.e., it stops forwarding messages). It resumes the protocol when it detects itself as being stationary, by joining the local cell. Notice that in this event, data stored by the node in the previous cell is replicated only at that cell. The data owner however, will also update its new location in the metadata of previously inserted objects (behaving as a passive replica for that content). While moving, nodes still process received periodic beacons, allowing them to keep communicating with the GHT.

4.5.6.4 Negative Acknowledgments

According a typical GHT interface, nodes are not individually addressable, i.e., we only send messages to specific geographic positions (that correspond to cells in our cell-based approach). Nonetheless, we support the sending of messages to a node in a *specific cell* (e.g., send a message to node *a* in cell 12).

To allow the upper layers to react to a node failure or migration from one cell to another, the routing layer replies with a NACK to a *message source*, when a message addressed to an individual node could not be delivered (because the node was not in the supposed cell).

4.5.6.5 Message Destination Aggregation

For messages that are to be delivered to multiple destinations (e.g., notifications), we optimized our routing scheme by only propagating a single message to those destinations, in what we call *message destination aggregation*.

A message is only duplicated when strictly required, which happens when the message's next hop for different destinations is not the same, as depicted in Figure 4.5. Thus, achieving a kind of tree-like routing, contributing to reduce the energy consumption and the occupancy of the wireless medium.

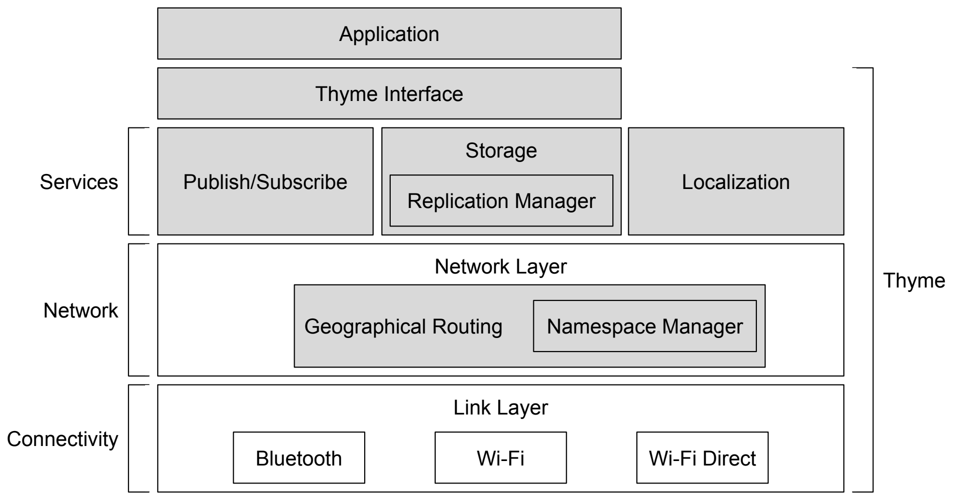


Figure 4.6: Thyme-DCS Android library architecture [46].

Naturally, this mechanism is more effective in sparsely populated scenarios (Figure 4.5a), as there are less possible paths where messages can be duplicated. Contrary, in densely populated scenarios, since there are more direct paths from any source to any other destination, this observation cannot be exploited so efficiently (Figure 4.5b).

4.5.7 Joining the System

A node joining the system waits a configurable amount of time, listening for other nodes' periodic beacons sent by its neighbors. If, during that time, it receives a beacon sent by a neighbor in its own cell, the sender of that beacon is used as an entry point. A join request is then exchanged, and the joining node receives all the cell state in a reply. If a maximum number of retries is reached, the node assumes it is alone in the cell, and starts operating normally, i.e., the cell was empty, and is now occupied as described before.

4.6 An Android Implementation

Both Thyme-LS and Thyme-DCS were implemented in the ns-3 network simulator [216] to allow large-scale experiments (see §4.8). Yet, to be able to experiment in real world scenarios, even though in a small scale, we address the application of Thyme to networks of Android mobile devices, implemented in the context of the M.Sc. of Cerqueira [46]. Here, we focus on the Thyme-DCS approach due to its range of applicability, since it copes with mobility and churn concerns. We apply it to real world networks of Android devices and use it in the development of a photo sharing application.

4.6.1 Architecture

Figure 4.6 depicts the multi-layer software stack that executes at each node. We present this architecture from a top-down perspective focusing on the challenges raised by

Table 4.4: Signature of the insert and subscribe operations in the Android library.

```
void insert (DataObject data, Collection<Tag> tags, byte[] description, OperationHandler opHandler)
void subscribe (TagExpression query, Time start, Time end, NotificationHandler notHandler, OperationHandler opHandler)
```

`data`	–	data object to store
`tags`	–	tags associated with the data object
`description`	–	description of the data object to store
`opHandler`	–	handler for handling the success or failure of the operation
`query`	–	tags relevant for the subscription
`start/end`	–	validity time interval for the subscription
`notHandler`	–	handler for the reception of notifications of data objects matching the given subscription

implementing Thyme in real world networks of mobile devices.

Thyme Interface. It offers the (asynchronous) operations presented in §3. The outcome of these operations must be dealt with through the implementation of specific handlers (i.e., callbacks). As an illustration, Table 4.4 presents the signatures of the insert and subscribe operations for our Java prototype.

Publish/Subscribe. Separated in client and server counterparts, this module manages the match between stored objects and subscriptions, and emits the necessary notifications. The client side manages insert and subscribe operation requests, as well as the reception of notifications. In turn, the server counterpart manages the matching (and storage) of data objects and subscriptions, having into account that each data object/subscription features a namespace identifier (see §4.6.2).

Storage. Also divided into client and server counterparts, this module manages the storage of data objects and their metadata, as well as of subscriptions. The internal Replication Manager sub-module manages the active and passive replication mechanisms (§4.5.2).

In this implementation, passive replication is built-in and cannot be disabled. However, active replication is optional: its default can be set to either active or inactive, and can be overridden in a per operation basis. For instance, data stored by the Publish/Subscribe module (i.e., data objects and subscriptions) always use active replication to ensure that such information is replicated inside the responsible cell. In turn, the dissemination of commercial advertisements may not require persistent storing. The actual active replication model may be injected in the Replication Manager, in order to support different strategies such as cell-wide replication, independently of the cell's population, or maintain a certain number of replicas, also independently of the cell's population.

Localization. Geographic routing requires nodes to be able to determine their own geographical position. For that purpose, we are currently resorting to the globally available GPS information. This option allowed the rapid prototyping of the Localization service. However, there are other possible solutions [206].

Network Communication. One of the fundamental differences between the ns-3 Thyme and our Android implementation is the *ad-hoc* communication, as this mechanism is not readily available in the targeted off-the-shelf Android devices. To circumvent this limitation, all network communication in our prototype makes use of a communication library [217] developed in the context of the Hyrax research project[4]. This library supports one-hop and multi-hop networks by using one or more wireless networking technologies. Currently, the following are supported: Wi-Fi, Wi-Fi Direct and Bluetooth. Regarding the API, the library supports both synchronous and asynchronous unicast, and scoped broadcast messaging.

We adapted this communication library to support the GPSR protocol [135], and support cell-level routing. Messages may be addressed to either a node or a cell. In the latter case, a random node of the target cell is chosen to either route the message to the next cell or to process the message itself, if the destination has been reached. Cell-wide communication is achieved via the scoped broadcast functionality.

4.6.2 Multiple Namespaces

The ns-3 version of Thyme was designed with a single grid in mind (§4.3.3), being the grid defined before the system starts. This grid works as a namespace or like a directory in a file system. However, this real world implementation allows for multiple (overlapping or non-overlapping) grids/namespaces, and provides a namespace discovery mechanism. This feature allows for a two-level naming hierarchy that was flat before. For instance, in the context of the photo sharing application (see §4.6.4), this feature enables the existence of multiple photo galleries shared by different users.

To cope with this demand, an application may manage multiple instances of Thyme, each bound to its own namespace. These instances present the previously described instance-agnostic Thyme interface, but embed an internal unique identifier that will be used by all modules of the software stack, ensuring the clear separation between the data of the multiple Thyme instances.

The creation of a new namespace requires the configuration of the geographical area to be covered by the associated Thyme instance, and the name to use when advertising the instance to the network. Currently, namespaces/grids have a rectangular shape, and their configuration requires a reference point and its length in all four cardinal directions. The dimension of each cell is computed automatically and depends on the networking technology in use. In the case of Bluetooth and Wi-Fi Direct, the dimension is computed from the technology's usual communication range. In the case of Wi-Fi, the size of the cell is set by a platform configuration parameter.

4.6.3 Handling Mobility

Device mobility impacts Thyme in several ways. First of all, it is necessary to know the device location, in order to: i) deliver notifications; ii) send replies to previously

[4]`http://hyrax.dcc.fc.up.pt`

issued requests; and iii) keep track of the whereabouts of passive replicas. Secondly, it is necessary to know in which cell a device is parked so that device may contribute to the cell's responsibilities, namely storing data objects and subscriptions.

Mobility is detected by sensing the device's accelerometer. Subsequently, the node will switch to *mobility* mode as soon as it leaves its cell, and will persist in such mode until it remains stationary for a configurable time period. While it is moving, a node will not work on behalf of any cell, but will process messages addressed to itself, such as notifications. To that end, as it moves across cells, the node will have to update its subscriptions' data with its new location (§4.5.4.3).

When the system locally detects that a node is no longer moving, if the final cell is not the same as the origin, the node discards all the (meta)data kept about the origin cell, updates its location in the system (namely with respect to the passive replicas it holds), and begins working on behalf on its new cell, replicating data and answering requests.

4.6.4 Shared Photo Gallery

A practical Thyme use case is a photo sharing application to be used at social events. Thus, as a case study, we developed the *shared photo gallery* application that allows users to share photos without requiring Internet access. The app can run on any device with Android 5.0 (Lollipop) or higher, without having *root* access, and works even in the absence of a communication infrastructure (when using Bluetooth or Wi-Fi Direct).

Users publish (or insert) photos with at least one tag and subscribe to the tags they are interested in, indicating a validity time frame for each subscription. This time frame may be unbounded in both ends, allowing for subscriptions to cover the event (or system) lifespan. Whenever a published photo matches one of the active subscriptions, a notification is sent with the photo's thumbnail (and a list of possible download locations). Upon reception of such notification, the user may choose to immediately start the download, postpone it, or discard the notification. Whatever the action, the user will be informed of its success or failure.

Figure 4.7 depicts some of the application's screens. In Figure 4.7a it is possible to identify four tabs:

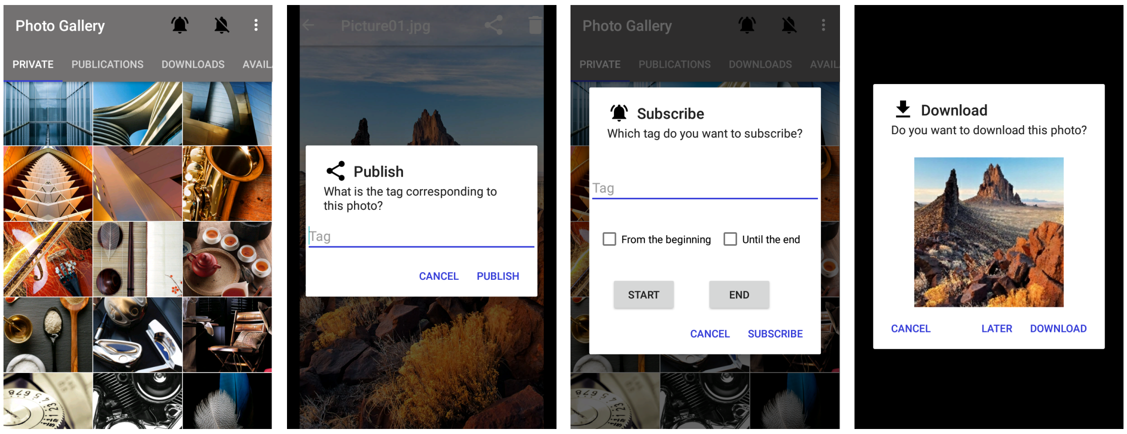
Private displays the private photos from the device's gallery, that can be published;

Publications displays the photos already published by the user;

Downloads shows the photos that were previously downloaded;

Available displays the photos whose download has been postponed.

Also, in this figure one can see the subscription and unsubscription buttons, represented by the bell symbols in the upper right corner. The other figures (Figs. 4.7b–4.7d) illustrate the processes of, respectively, publishing a photo, issuing a subscription, and handling the reception of a notification.



What is the tag corresponding to this photo?

Which tag do you want to subscribe?

Do you want to download this photo?

(a) Main screen. (b) Publish a photo. (c) Subscribe a tag. (d) Download a photo.

Figure 4.7: Shared photo gallery Android application.

The application may interact with more than one gallery. Users may thus search and connect to active galleries on neighboring devices[5] or create their own galleries. To navigate between galleries the user has simply to access the menu (i.e., the three vertical dots) in the upper right corner and select the *Switch gallery* option, which will lead to a (selectable) list with the available galleries.

4.7 Analytical Study

We now compare our approaches using a simple analytical model to derive approximate formulas for communication costs and operations complexity. In the following, we use the asymptotic costs of $O(n)$ message transmissions for floods and $O(\sqrt{n})$ for point-to-point routing, where n is the number of nodes in the system [212]. However, since in THYME-DCS we cluster nodes into cells, point-to-point routing still costs $O(\sqrt{n})$ but, here, n becomes the *number of cells* in the system.

As a baseline for comparison, we devise an additional approach, THYME-ES, using the client/server model and based on external, centralized storage. Storage is external in the sense that it does not belong to the nodes forming the network, i.e., it belongs to a different (server) component, known a priori by every node in the system. Objects, their metadata, and subscriptions are stored in external storage, and every operation is sent to that server to be processed (and replied back). Naturally, this server component is a single point of failure in the system, but can use any known techniques from the literature to address this issue (e.g., fail-over, or state machine replication [32]).

4.7.1 Time Complexity

The operations (average) time complexity is as shown in Table 4.5. Delete and unsubscribe are the inverse operations of insert and subscribe, respectively. Despite their respective

[5]Access control and security is mandatory in this environment but falls outside the scope of this work.

Table 4.5: Thyme operations time complexity.

	ES	**LS**	**DCS**
Insert/Delete	$O(\sqrt{n})$	$O(1)$	$O(\sqrt{n})$
Retrieve	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$
Subscribe/Unsubscribe	$O(\sqrt{n})$	$O(n)$	$O(\sqrt{n})$

messages carrying slightly different information, in terms of complexity, they exhibit the same behavior.

Since ES and DCS use point-to-point communication for every operation, their complexities are the same. However, while in ES, the majority of the work is executed by the external component, in DCS, the work is spread among the cells, i.e., the system nodes. Also, take into account that by clustering nodes into cells, the point-to-point routing has the potential to be more efficient. In turn, LS trades linear (un)subscribe operations for constant inserts and deletes. In all approaches, retrieve operations are executed using point-to-point communication.

4.7.2 Space Complexity

Regarding space complexity, ES exhibits an extreme behavior, because the external component has to store every piece of data of the system (i.e., objects, their metadata, and subscriptions).

In turn, LS sits in the middle of the spectrum, by storing objects only in their owners' storage, but fully replicating every subscription.

Lastly, DCS spreads both the storage of objects, metadata, and subscriptions among its cells/nodes, through hashing. In terms of storage, the LS and DCS approaches are not directly comparable. However, they both reside in the middle of the spectrum, spreading (different) parts of the system data among the nodes.

4.7.3 Communication Costs

The communication cost structure for each approach is described by several parameters. Let D_i denote the total number of stored objects; D_r denote the number of retrieved objects; S denote the total number of issued subscriptions; and D_s denote the number of matching objects (i.e., the total number of notifications). For DCS, c is the (average) number of nodes in a cell.

We compare costs using approximations for both the total number of sent messages in the network (taking into account multi-hop routing), and the number of messages sent by a hot-spot (i.e., the *maximal* number of messages sent by *any* particular node). In this comparison, we only address the insert, retrieve, and subscribe operations, and the respective notifications. The delete and unsubscribe operations are analogous to insert

Table 4.6: Thyme operations communication costs.

	ES	**LS**	**DCS**
Insert	$2D_i\sqrt{n}$	$\emptyset$	$2D_i\sqrt{n} + 2D_i \cdot c$
	$+$	$+$	$+$
Retrieve	$2D_r\sqrt{n}$	$2D_r\sqrt{n}$	$3D_r\sqrt{n} + D_r \cdot c$
	$+$	$+$	$+$
Subscribe	$2S\sqrt{n}$	$S \cdot n$	$2S\sqrt{n} + S \cdot c$
	$+$	$+$	$+$
Notification	$D_s\sqrt{n}$	$D_s\sqrt{n}$	$D_s\sqrt{n}$
Hot-spot	$D_i + D_r + S + D_s$	$D_r + S + D_s$	$D_i + D_r + S + D_s$

and subscribe, respectively. With this setup, the approximate communication costs (total and hot-spot) are as shown in Table 4.6.

The formulas for ES are derived from the observation that every operation is sent to the central component to be processed, and a response is sent back to the requester. Thus, for every operation, the communication cost (of a point-to-point message) is multiplied by two. The exception are notifications that only require a message sent in one direction (from the central component to the subscriber). Naturally, the hot-spot is going to be the central component, which has to process every received message and reply accordingly. Hence, the hot-spot communication cost is the sum of all the received messages (that have to be replied) and the sent notifications.

For LS, the formulas are deduced from the facts that insert operations are local and do not require communication, while subscriptions are flooded through the entire system (§4.4). Both the retrieve operation and notifications follow the same logic as the previous approach.

Lastly, the formulas for DCS are inferred taking into account its cell-based GHT approach. The insert operation requires the same steps as ES—two point-to-point messages (to the broker cell and back to the requester). However, in each of these two steps there is a scoped dissemination of the corresponding messages in the local cells (i.e., the dissemination of the metadata in the broker cell, and the dissemination of the data object in the data owner's cell; see §4.5.2.1). Next, the retrieve operation requires the same steps as the other two approaches (two point-to-point messages), with an additional point-to-point message for setting a new passive replica (§4.5.2.2) and the corresponding scoped dissemination in the broker cell. The subscribe operation follows the same logic as the insert. However, it only has one scoped dissemination of the subscription in the broker cell. Notifications follow the same logic as the previous approaches (sending a point-to-point message directly to the notification receiver).

Here, we assume a simple scenario where inserted data objects have only one tag, and subscriptions also have only one conjunction key. In more elaborate scenarios, the (possibly variable) number of tags will have impact in the number of messages required for some operations.

Now, we can conclude that the total message count in LS grows faster (linearly with n) than in ES and DCS. Another important conclusion is that, if $D_i \gg S$, then LS has significantly lower message count than the other two approaches. This comes from the fact that LS insert (and delete) operations execute with no communication. Naturally, if we invert that condition, ES and DCS will exhibit lower message counts than LS.

Once again, ES and DCS exhibit a similar behavior in terms of overall communication costs. However, DCS presents slightly higher costs in almost every operation, thus they have intrinsically different performance behaviors. These higher costs come from the replication mechanisms employed by DCS (§4.5.2). In an insert operation, besides the normal request/reply messages, by applying active replication (§4.5.2.1), both the object data and metadata are (actively) replicated in the owner's and responsible cells, respectively. In the retrieve operation, after obtaining the object data, the requester node passively replicates that object, thus needs to update the replication list in the object metadata (§4.5.2.3). Regarding subscribe operations, subscriptions are also actively replicated in their responsible cells.

With these extra mechanisms, naturally many operations in DCS have a slightly higher communication cost. However, while ES has an external, central component acting as a server (that is also a single point of failure of the system) and storing all the system's data, DCS spreads that load among its cells/nodes. Thus, in ES the hot-spot cost is the actual cost paid by the external server. On the other hand, the hot-spot cost in DCS is not actually paid by a single node, because that cost is shared among the different cells (and among the nodes of each cell). Even if there is only one cell, this work will be (randomly) balanced between the nodes inside it.

We can also look at these costs as the amount of work a device has to do on behalf of the system. Specific to the DCS approach, the handling of messages related to the five operations grows linearly with the number of such operations, and does not depend on the number of nodes per cell. That is, n operations require each node on the cell responsible for the target tag to process on average n messages: one node receives the initial message and then broadcasts it to its cell neighbors. The same happens on each retrieve operation: a message is sent to the target cell to indicate the existence of a new passive replica, and this information is then broadcasted within the cell. Regarding active replication, the use of such mechanism implies one broadcast on the owner's cell for each operation. So, with active replication each node in such cells processes on average one message per operation.

The only type of message that depends on the number of nodes per cell is the one concerning notifications. As depicted in Figure 4.8, the more populated a cell is, the less work each node has to do. These only require the intervention of one node per cell, the one checking the match between an object and a subscription, and sending the notification to the corresponding subscriber. This work is also load balanced, because the object-subscription matching is not performed by the same node inside a cell. It is distributed randomly among the cell nodes (during message routing).

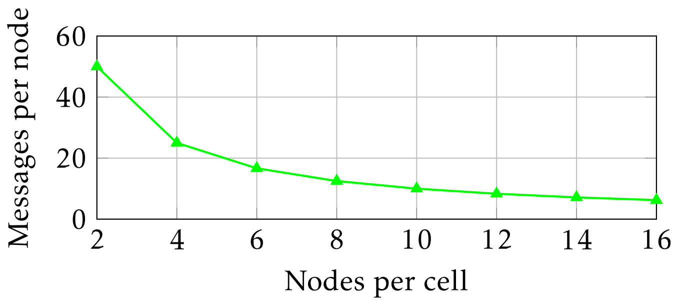


Figure 4.8: Average number of messages sent per node in Thyme-DCS, for 100 notifications processing.

4.7.4 Discussion

In the end, if the number of inserted data objects is larger than the system size and the number of subscriptions, Thyme-LS may be preferable. However, this approach does not address data persistence in case of node failure.

On the contrary, Thyme-DCS addresses data persistence through replication. It should be preferable in cases when the network is large compared to the number of stored objects, being more worthwhile in densely populated scenarios.

4.8 Evaluation Through Simulation

Our experimental evaluation is divided in two parts: simulation and real world experiments. First, we use a network simulator (ns-3 [216]) to experiment our proposal in large-scale scenarios, and implemented our two approaches: Thyme-LS and Thyme-DCS. Secondly, we implemented the Thyme-DCS approach as an Android library, and developed a proof-of-concept photo sharing application on top, allowing experiments in a small scale scenario using real mobile devices.

This part of our evaluation focus on the simulation experiments and seeks to answer the following questions:

1. Which are the trade-offs provided by each approach of Thyme?
2. How does each approach deals with churn?
3. How do they react to node mobility?

Each data point reports the average of five randomly generated network topologies, each independently run three times, thus making a total of 15 runs per data point. As a baseline for comparison, we used the centralized approach, Thyme-ES, described at the beginning of §4.7.

The metrics used in this section to answer the previously defined questions are: amount of generated traffic (in bytes and number of packets), and operations' latency and success ratio. All these metrics allow the comparative analysis of the behavior of both Thyme approaches. Since we are addressing resource-constrained mobile devices, the lower the generated traffic and the operations' latency the better, because this has direct implications in the devices' battery usage. In the end, the best approach is the one able to achieve the lowest latency and generated traffic while producing high operations' success ratios.

4.8.1 Implementation

We use ns-3.27 and nodes communicate through 802.11 Wi-Fi ad-hoc (using UDP). Both Thyme-ES and Thyme-LS use DSDV [200] as their routing protocol. From a practical standpoint, it was the routing protocol that resulted in more timely simulation runs[6].

In Thyme-DCS, when a cell becomes empty or populated, a state transfer needs to happen between cells (§4.5.6.2). Currently, we do not implement such mechanism, thus, in our experiments, cell structure is static (i.e., populated and empty cells will remain as such throughout the experiments). This poses some limitations regarding node mobility and churn in Thyme-DCS. As such, nodes may move freely inside a cell, but may only leave a cell if it remains populated afterwards. Also, nodes may only migrate to previously populated cells.

To recover from lost messages, all approaches employ a retransmission mechanism. After a configurable amount of time has passed without receiving the expected replies, the operation is retried. If a (configurable) maximum number of retries is reached, the operation fails with a timeout error code.

Our implementation of Thyme is available at `https://bitbucket.org/hyrax-nova/thyme-ns3`, jointly with the trace files used in the simulation experiments.

4.8.2 Setup and Methodology

Unless stated otherwise, all parameters were left with the simulator's default values. We used Wi-Fi 802.11g configured with a constant rate manager and a data rate of 6 Mbps. The RTS/CTS threshold was configured to 1500 bytes.

In order to mimic a realistic scenario, we emulate an application similar to an online social network on top of Thyme (akin to Twitter), e.g., that could be used by fans watching matches in fan zones set up for the 2018 FIFA World Cup.

Trace files were generated with all the operations to be issued during a simulation run. For that, we crawled tweets issued during the 2016 UEFA European Championship final, between Portugal and France[7]. Tweets were used as data objects, where: the tweet id was used as the object identifier; the text was used as the object data; the timestamp was used as the object insertion time; and the hashtags were used as the object tags. The

[6]Other protocols, such as OLSR, AODV, or BATMAN, resulted in simulations with a large number of nodes to take an impractically long amount of time to complete.

[7]Using the code in `https://github.com/Jefferson-Henrique/GetOldTweets-python`

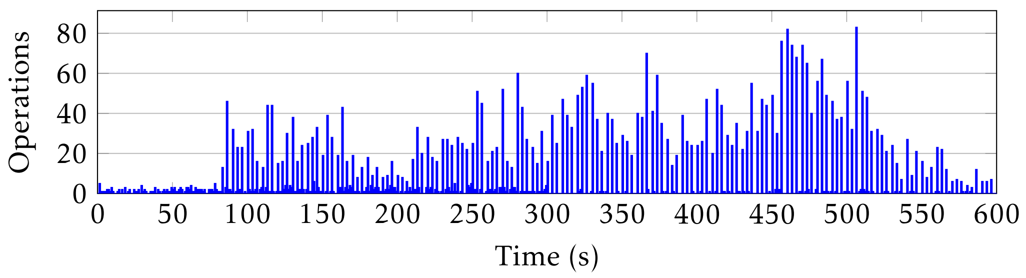


Figure 4.9: Distribution of operations over time in a trace.

top-k most active users were chosen, and every other operation was generated from that, using exponential distributions configured with different λ values (i.e., rates).

Subscriptions were generated taking into account the tags of the inserted objects, and the top 60% of the most popular tags were used for the subscriptions' queries (for simplicity sake, each subscription subscribed to one tag chosen uniformly at random). Subscriptions were generated in two forms: time independent ($ts^s = ts^e = \perp$); and in the future ($ts^s = now$ and $ts^e = \perp$). Time independent subscriptions where generated with a probability of 60%. During the first half of the game, subscriptions were generated with a rate of three operations per user per hour, and reduced to one per user per hour for the remainder of the event.

Delete and unsubscribe operations, which are expected to be rare, were generated with a rate of 0.5 and 0.2 operations per user per hour, respectively, only during the second half of the game.

We crawled a total of three hours, starting at 20:00 2016-07-10. To make the simulation execution more lively (and to reduce the simulation total time), we compressed the three hours into ten minutes of simulated time. Since we use real tweets for trace generation, the distribution of operations in a trace file is irregular, with occasional spikes and void moments. Figure 4.9 depicts an example of the distribution of operations in a trace file over time (for 100 users). The trace files used in this section are publicly available in our code repository[8].

The simulation area has a rectangular shape, where nodes were places uniformly at random, to mimic many of the venues we are targeting, like concert halls. For Thyme-DCS, cell size is 40×40 meters, which entails a radio range of ±113 meters (roughly the range in our simulated Wi-Fi setting). In all experiments, we had an average density of two nodes per cell, and used the simulation areas as mentioned in Table 4.7 (naturally, the last two rows only concern Thyme-DCS).

Proactive routing protocols, like DSDV, require time to converge. Thus, in our simulations, the application running on the nodes only started after 30 seconds. Nodes randomly joined the system in the next 30 seconds, and operations started being issued

[8]https://bitbucket.org/hyrax-nova/thyme-ns3/src/master/scripts/traces/files/

Table 4.7: Simulation area according to the number of nodes.

Num. nodes	16	36	64	100	144	196
Area (meters)	160×80	240×120	320×160	400×200	480×240	560×280
Cells	4×2	6×3	8×4	10×5	12×6	14×7
Total cells	8	18	32	50	72	98

only after that. At the end of the simulation, nodes only shutdown 60 seconds after operations stopped being issued. Thus, the total simulation time was 720 seconds. All Thyme approaches executed the same traces and used the same methodology.

Since notifications are "operations" not directly triggered by the users, we use *recall* as a measure of success. This tells us how many relevant items/objects are selected (or how complete the results are), and is computed by $\frac{True\ positives}{Relevant\ items}$ (following the concepts defined in Figure 4.10). However, here, we use the number of matching objects for a perfect execution of the trace, where operations *always* succeed and are executed *instantly*. Thus, take into account that if some operation fails, most likely the number of actual achieved notifications will not match the expected. For instance, if an insertion fails, all the subscriptions matching that object will not be matched against it, and notifications are reported as lost (i.e., false negatives). Thus, achieving 100% recall is practically impossible for this comparison criterion.

In turn, we do not refer to precision, because it is always 100%. This metric tells us how many selected items/objects are relevant (or how useful the results are), and is computed by $\frac{True\ positives}{Selected\ items}$. Referring to Figure 4.10, in our case, Thyme's matching procedure may lead to false negatives, but will never return false positives. That is, some objects may not be matched against a subscription, but all those that are and match the query, are relevant results.

4.8.3 Results

We now present the achieved results for three distinct scenarios, ranging from totally stable nodes to scenarios with faulty or mobile nodes.

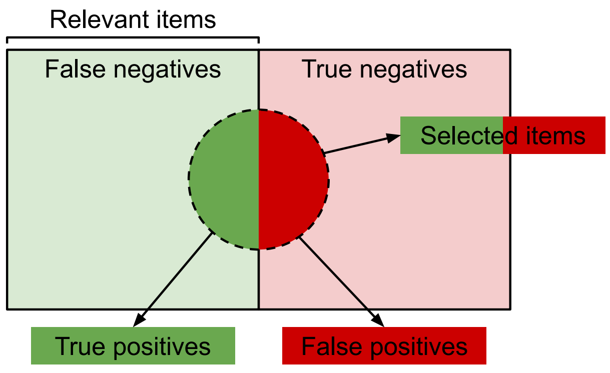
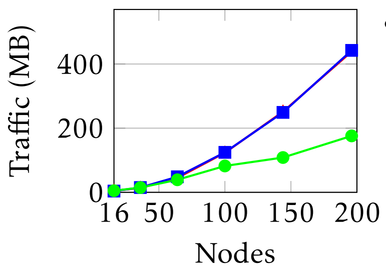
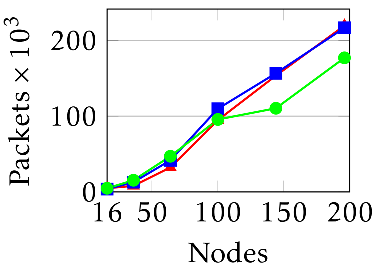


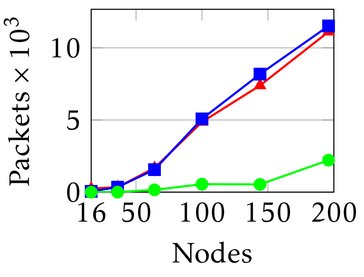
Figure 4.10: Precision and recall.



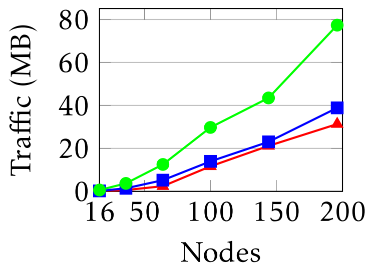
(a) Total transmitted traffic.



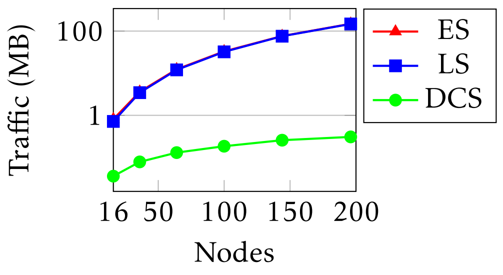
(b) Total retransmissions.



(c) Total failed transmissions.



(d) Total forwarded traffic.



(e) Routing control traffic.

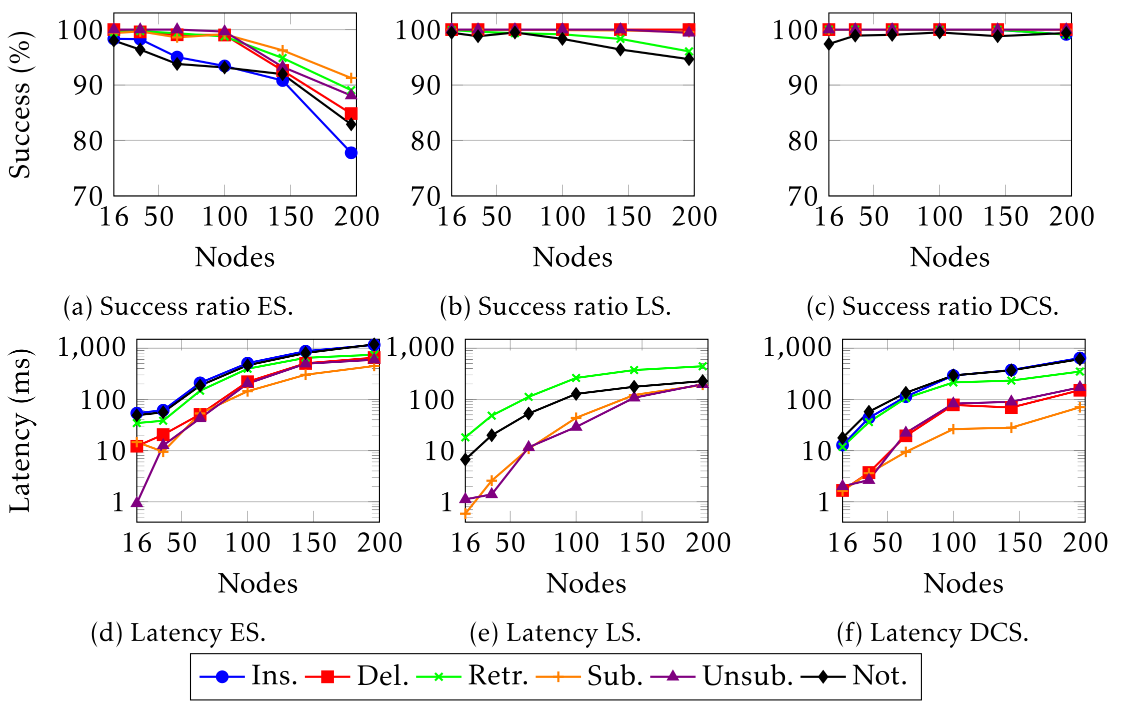
Figure 4.11: Thyme lower layers metrics (static scenario).

4.8.3.1 Static and Stable Nodes

In Figure 4.11, we can observe the impact that each approach of Thyme has on the lower layers of the network stack (and helps answering question 1). Figure 4.11a reports the total traffic transmitted by all nodes (at the physical layer—PHY), during the simulation. ES and LS overlap and both exhibit quite an overhead. With 196 nodes, they report more than 2× the transmitted traffic of DCS. Energy is a valuable resource when targeting mobile devices. Thus, looking at these values in an energy perspective, ES and LS will spend twice the energy to do roughly the same work as DCS.

In turn, Figures 4.11b and 4.11c depict values reported by the link layer—MAC. The former shows the total number of retransmitted packets, and the latter shows the total number of packets that exceeded the maximum number of retransmission attempts. The standard IEEE 802.11 Wi-Fi MAC layer implements CSMA/CA and a per hop retransmission mechanism. Thus, in some way, these figures depict the interference level observed in each approach while operating. Both ES and LS require many more retransmissions than DCS to overcome the loss of messages that is inevitable in a wireless communication medium. This can be explained by the amount of traffic generated by those two approaches. Usually, the more traffic is generated, the larger is the probability of collisions in the wireless medium, and thus more transmissions have to be retried (creating a snowball effect).

Next, Figure 4.11d depicts the total traffic forwarded by all the nodes in the system (at the network layer—IP). In some sense, this reports the amount of work nodes have to do on behalf of the system. In this case, DCS forwards more traffic because its messages are



(a) Success ratio ES. (b) Success ratio LS. (c) Success ratio DCS.

(d) Latency ES. (e) Latency LS. (f) Latency DCS.

Figure 4.12: Thyme application-level metrics (static scenario).

forwarded through longer routes than ES and LS (that use DSDV). This is even more exacerbated by some peculiarities of the routing protocol used by DCS (§4.5.6.1). For instance, the fact that some messages may need to loop around voids in the network (§4.5.6.2), while DSDV computes shortest paths to every node.

Figure 4.11e shows the total amount of control information the routing protocols transmit. Both ES and LS use DSDV, a proactive routing protocol, whereas DCS uses a geographic routing protocol (§4.5.6.1). While DSDV needs to exchange bulky routing tables to compute the shortest paths to every other node in the network, the geographic routing used by DCS routes messages using only local information (nodes only exchange very small periodic control beacons). However, messages may be routed through longer routes in geographic routing. With 196 nodes, a quarter of all the transmitted traffic of ES and LS was control traffic (notice the logarithmic scale in the y axis). These two last figures (Figure 4.11d and 4.11e) show a clear trade-off. As more control traffic is exchanged, the routing protocols can achieve better routing paths and with that reduce the amount of forwarded traffic. However, that control traffic can correspond to a large percentage of the total network traffic (increasing the total amount of collisions).

Figure 4.12 depicts application-level metrics, such as operations success ratio and latency. Regarding operations success, we verify that DCS is above 99%, except for notifications that fluctuate a little bit and have a success ratio as low as 95% (Figure 4.12c). LS also reports high success ratio (Figure 4.12b). Since insert and delete operations are executed locally, they always succeed. Subscribe and unsubscribe operations have above 99% success. Only retrieve operations and notifications have a very small reduction as the system grows, having 96% and 95% success, respectively, with 196 nodes. For ES (Figure 4.12a), we see a slight decrease in the success ratio as the system grows, having as low as 78% success with 196 nodes. In every approach, notifications are a type of message that does not employ an application-level retransmission mechanism, thus they are more susceptible to interferences.

Regarding operations latency (Figures 4.12d–4.12f), we can see that for a small number of nodes all approaches behave similarly, with ES having slightly higher latencies. As the number of nodes increases, accompanied by increased interferences (Figure 4.11c), we verify that latencies also increase. This is caused by the need for more retransmissions. However, notifications have lower latency in LS, because the geographic routing of DCS cannot compete with the shortest paths of DSDV. Thus, showing the advantage of calculating shortest paths. On the other hand, retrieve operations in DCS have a slightly lower latency, because DCS causes overall less interferences and it employs a location-aware strategy when retrieving data (§4.5.5). In ES, the decrease in success ratio is accompanied by an increase in operation latency. This comes from the fact that the majority of operation failures happen due to timeout. Since operations have to be retried several times, naturally latency increases. Overall, timeouts may indicate a congested network, where operations consistently have to be retried several times. Also, note that ns-3 does not account for processing time (i.e., time spent executing the protocols' logic). Otherwise, it would exacerbate ES latencies even more, since it has a central coordination point that inevitably will become the system's principal bottleneck.

In summary, Figure 4.11 shows that in DCS nodes transmit much less traffic that in both LS and ES. This comes at the cost of latency, when compared to LS, as shown in Figure 4.12. The centralized approach has the worst behavior when the size of the system grows, with a decreasing success ratio and latency much higher that both DCS and LS. These observations come from the fact that both ES and LS use DSDV as the underlying routing protocol. This protocol computes shortest paths to every other node in the network, and to maintain its routing tables up-to-date that information is distributed between nodes by sending full dumps infrequently and smaller incremental updates more frequently, which still represent a large overhead with respect to transmitted data. On the other hand, DCS uses GPSR as its routing protocol, which uses only local information for routing. Thus, they represent a design trade-off: the more control information is transmitted, the better routing decisions can be made.

4.8.3.2 Static but Failing Nodes

In our target scenarios, mobile devices may experience poor connectivity and/or low battery, thus these devices may fail and leave the network. As such, concerning churn, i.e., the ingress and egress of nodes in the system, we experiment with two different scenarios. We show the impact of nodes leaving the system definitely, e.g., nodes crashing. Secondly, we show the impact of nodes with intermittent failures, thus entering and leaving the

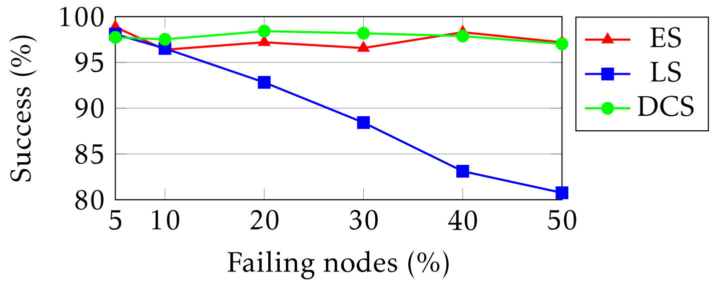


Figure 4.13: Thyme notification success ratio (permanent failures, 100 nodes).

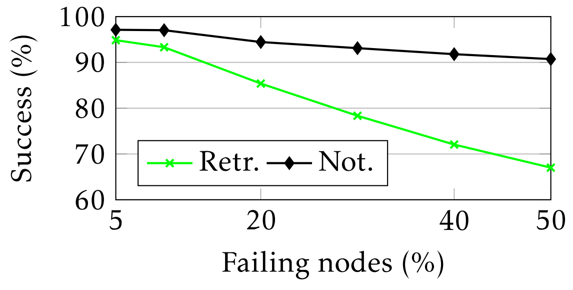
system multiple times throughout the simulation. These scenarios allow to evaluate aspects regarding data availability and persistence in the presence of failures (and help us answer question 2).

Permanent Failures. In this scenario, from the same trace files as before, we generated new ones where nodes are *either* publishers or subscribers, and publishers choose a random instant (between 200 and 300 seconds of the simulation, i.e., around the middle of the simulation) to leave the system abruptly.

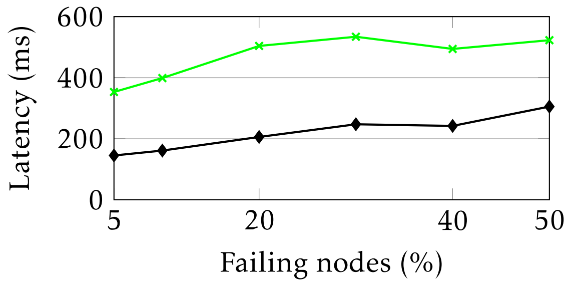
In LS, insertions are executed locally, thus not requiring communication. However, because only the object owner stores that data, if that node fails, all the data it stores will disappear with it. Figure 4.13 shows exactly that. In LS, as more nodes with relevant data fail and leave the system, more the success ratio decreases because the matching between subscriptions and objects is not detected. Since DCS employs replication mechanisms (§4.5.2), even when object owners leave the system, matching still occurs. ES is not affected simply because all the system data is stored in external storage. As long as that server component does not fail, even if nodes do, data will always be available.

Transient Failures. In this scenario, using the same trace files as in §4.8.3.1, randomly selected nodes alternate between the on and off states, during 120 and 60 seconds respectively. Nodes have a 75% probability of changing to the opposite state, otherwise they stay in the same state for an equal period.

With nodes entering and leaving the system frequently, retrieve operations and notifications are the ones that can be more affected, specially in the LS approach. Figure 4.14 presents the success ratio and latency of these two operations for the LS approach. Regarding the other two approaches, since DCS employs replication mechanisms, it is little affected by the intermittent churn, with operation success ratio well above 90%, and latencies consistently between 150–300ms. In turn, due to its central server component, ES is also little affected by the intermittent churn, with operation success ratio above 80%, and slightly higher latencies than DCS, between 400–600ms. However, LS suffers from low success ratio in the retrieve operation (Figure 4.14a). Although the matching among some objects and subscriptions is detected, and some notifications are sent, when a node



(a) Success ratio.



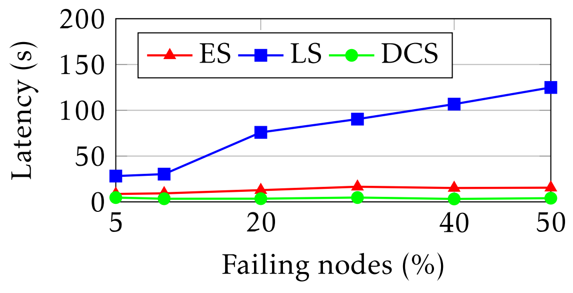
(b) Latency.

Figure 4.14: Application metrics for Thyme-LS (transient failures, 100 nodes).

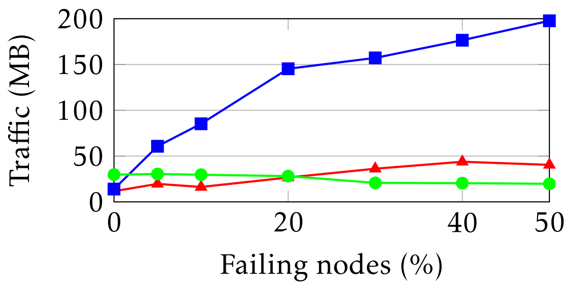
tries to retrieve an object, as the amount of failing nodes increases, the probability of the data owner being off also increases. This is also accompanied by an increase in the latency of notifications (Figure 4.14b). Note that, in LS, the matching between a node's stored objects and subscriptions that were issued when the node was off have to wait for the node to switch state and join the system (§4.4).

When joining, a node receives the subscriptions issued by all the other nodes previous to it entering the system (from its neighbor nodes). Then, from all the received subscriptions, the joining node finds those (new subscriptions) of which it was unaware and checks if it has matching objects. Figure 4.15a corroborates this. The maximum latency for DCS and ES notifications stays stable as the percentage of failing nodes increases. But, in LS, the maximum latency for a notification increases from approximately 25 to 100 seconds when the percentage of failing nodes increases from 5% to 40%.

Additionally, Figure 4.15b shows a byproduct of the retrieve operation low success ratio. With no churn, DCS forwards more traffic because its insert and delete operations require communication. However, with this kind of intermittent churn, we observe that LS forwards much more traffic than DCS. This is due to the fact that retrieve operations are retried (and fail) several times. Also, this entering and leaving of nodes from the network causes routing tables to become out of date, and thus need to exchange control information much more frequently.

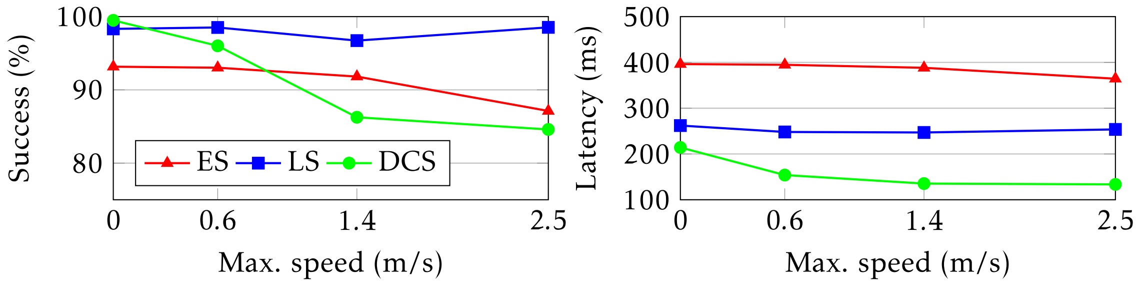


(a) Maximum notification latency.



(b) Total forwarded traffic.

Figure 4.15: Transient failures scenario in Thyme (100 nodes).



(a) Notifications success ratio.

(b) Retrieve latency.

Figure 4.16: Mobile scenario in Thyme (100 nodes, pause 120 seconds).

4.8.3.3 Mobile but Stable Nodes

In this scenario, we use the same trace files as in §4.8.3.1, however nodes are able to move, thus helping us answer question 3. When moving, nodes use the random waypoint (RWP) mobility model, which interleaves pauses with movement. However, we argue that the plain RWP mobility model does not quite mimic the movement pattern people have in the kind of events we target. For instance, in a music concert (or other cultural events), people do not move constantly. In fact, they do not move much during most of the time, except in intermissions. To make it better resemble our target scenarios, we made an adaptation: every time a node is about to move, it tosses a coin do decide whether to move or not. If not, the node continues in a pause moment. In this scenario, 60% of nodes are mobile, and have a moving probability of 80%.

Figure 4.16a shows a small caveat of DCS: increasing node speed lowers the notifications success ratio. We argue this happens because every node inside a cell is supposed to have the same state and work collaboratively as one. But, the intra-cell communication primitive is the unreliable one-hop broadcast. Thus, nodes inside a cell may not receive the same messages. The added mobility may create even more entropy in the cell state.

In turn, Figure 4.16b presents a byproduct of the location-aware retrieval strategy used by DCS (§4.5.5). While, ES and LS are required to retrieve data from one specific location, DCS allows having different replicas for retrieval at its disposal. Additionally, it can choose the location closer to the requester (having the possibility of lowering latency).

4.9 Evaluation Through Real Devices

This second part of our evaluation (executed in the context of the M.Sc. of Cerqueira [46]) has the main goal of assessing the functionality of our Thyme Android library and of the proof-of-concept application. For this, we seek to answer the following questions:

1. Does the implemented Android Thyme-DCS library behaves as expected?
2. What is the behavior of our Android application in terms of operations' latency?
3. What about in terms of energy consumption?

Each data point reports the average of five independent runs.

The metrics used in this section to answer the previous questions are: operations' latency and energy usage. These metrics allow the analysis of each operation's behavior in the context of the Thyme-DCS approach implemented in real Android devices. Once again, since we are addressing resource-constrained mobile devices, the lower the operations' latency and the energy usage the better (having into account that these two metrics can have a direct correlation between them).

4.9.1 Implementation

Both the Thyme-DCS library and the Shared Photo Gallery application were developed for devices with Android 5.0 (Lollipop) or higher, with no root access required.

In our Java prototype, we also do not implement the state transfer mechanism when cells become empty or populated (§4.5.6.2). So, cells must remain populated or empty throughout the experiment. Here, we also did not implement some optimizations like NACKs (§4.5.6.4) and message destination aggregation (§4.5.6.5). Similarly to the ns-3 implementation, here operations also employ a retransmission mechanism (§4.8.1).

4.9.2 Setup and Methodology

We conducted a series of experiments with different scenarios trying to simulate some possible realistic use cases. Here, we test all the features provided by the Thyme-DCS library and used by the photo sharing application: devices insert and delete photos, subscribe (and unsubscribe) to tags in the past and future, receive notifications, and retrieve available photos. Each device has a randomly assigned role (publisher or subscriber), and operations are executed in a closed loop. We used a custom profiler to collect various metrics during these experiments aiming to analyze several performance indicators, such as latency and energy consumption.

Unless stated otherwise, experiments were conducted using images with 35 bytes in size. This allow us to measure the operations latency and overheads taking only into account the data generated by the system. Naturally, as the size of the inserted data grows, latency and overheads also grow proportionally with respect to the available bandwidth.

Experiments were conducted in a network of up to 32 Android mobile devices, connected to a Wi-Fi access point. The tests up to 16 devices were performed exclusively with Nexus 9 devices, while the 32 devices testbed used every type of device referred in Table 4.8. In all experiments, we used two cells and devices were divided equally between the two. We also used only one namespace for all devices.

4.9.3 Results

We now present the achieved results for the collected metrics. The error bar depicted in the plots indicates the standard deviation.

Table 4.8: Mobile devices specifications for Thyme-DCS Android experiments.

	HTC Nexus 9	Motorola Nexus 6	LG Nexus 5X	Motorola Moto G (2nd gen.)
CPU	Dual-core 2.3 GHz	Quad-core 2.7 GHz	Hexa-core 4x1.4 GHz 2x1.8 GHz	Quad-core 1.2 GHz
RAM	2 GB	3 GB	2 GB	1 GB
Storage	16 GB	32 GB	16 GB	8 GB
Battery	Li-Po 6700 mAh	Li-Po 3220 mAh	Li-Po 2700 mAh	Li-Ion 2070 mAh
OS	Android 7.1.1	Android 7.1.1	Android 7.1.1	Android 7.1.1
Wi-Fi	802.11 a/b/g/n/ac	802.11 a/b/g/n/ac	802.11a/b/g/n/ac	802.11b/g/n

4.9.3.1 Functionality

With all the uses cases, we conducted experiments in order to test all the operations and features of the Thyme Android library. After extensive testing, we verified that all operations were performed successfully, including those involving the retrieval of data inserted by devices that had left the system. The use cases with churn allowed us to test the operations in scenarios with device or communication failures.

In the end, throughout all the experiments, operations were always completed successfully, proving the data persistence in the system. It should be noted that the retrieve operation's success, in the cases with churn, is guaranteed at the expense of an increase in the operation latency, i.e., if a device that left the system is selected or if the message is lost, the operation will be retried after a timeout.

4.9.3.2 Latency

Operations latency is an important measurement for assessing an application's usability. For that purpose, we measured the latency of all five operations (during the execution of the use cases) and present the average.

Figure 4.17 depicts the latency of those operations, varying the number of mobile devices in the network. It represents the time elapsed from the moment the action was triggered by the user in the application interface, until the reception of the operation's success reply. In general, we can consider that all the operations show acceptable response times (around 0.2 seconds on average). Insert is the operation that may take longer, depending on the size of the thumbnail sent in the metadata, which was kept particularly

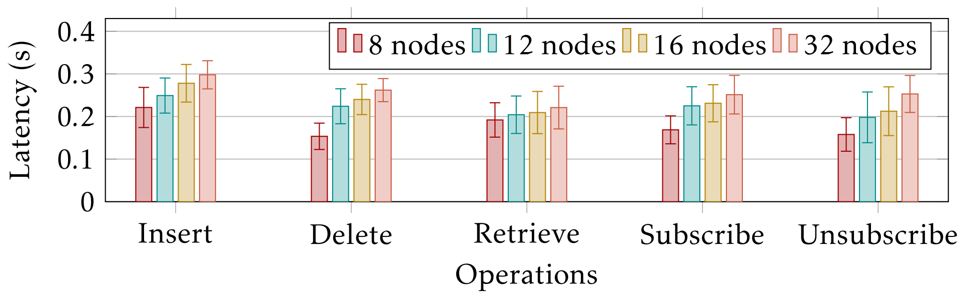


Figure 4.17: Operations latency in Thyme-DCS Android [46].

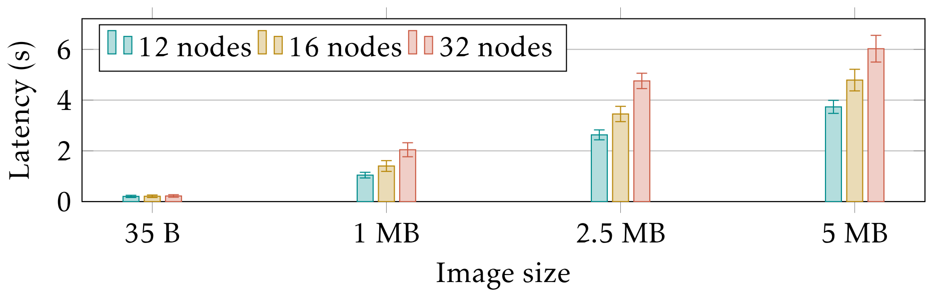


Figure 4.18: Retrieve operation latency varying image size in Thyme-DCS Android [46].

small (35 bytes) in our experiments. The experiment confirms that increasing the number of nodes also increases the network traffic, which in turn increases interference, reduces the available bandwidth for each device, and impacts the latency of the operations. Even so, all results are kept under 0.3 seconds, which is perfectly acceptable, with all the operations executed in quasi-real-time and ensuring a good user experience.

Regarding the retrieve operation, the latency depends on the size of the data item to be obtained, as shown in Figure 4.18. An image with 5 MB in a network of 32 devices, takes about 6 seconds, which is acceptable considering that the operation runs in the background and the user may keep on using this or other application on the device.

In conclusion, the implementation of Thyme and the developed application meet our expectations, presenting good response times, which guarantees a good interactive experience to the user.

4.9.3.3 Energy Consumption

When it concerns mobile devices, energy consumption is a determining factor, since these devices are battery-constrained. In order to fully evaluate the energy consumption of our application, we used the aforementioned use cases and measured the energy consumption, breaking it down into three parts: i) when issuing an operation; ii) when processing an operation request; and iii) in the maintenance of the virtual node, i.e., update the state after a new request is received for the cell.

Battery consumption was measured exclusively on the Nexus 9 devices, to avoid variations in measurements that could occur if different devices were used. The battery measurements were done automatically via a module that uses the *BatteryManager* class provided by the Android OS[9]. After a first analysis of the results, we concluded that the energy consumption values did not depend on the number of devices in the network, since we verified a marginal variance. Thus, we present these results as an average, independent of the number of devices in the network.

Issuing a Request. Figure 4.19 presents the energy consumption when issuing each operation. From this plot, we can verify that every operation uses very little energy. For

[9] https://developer.android.com/reference/android/os/BatteryManager

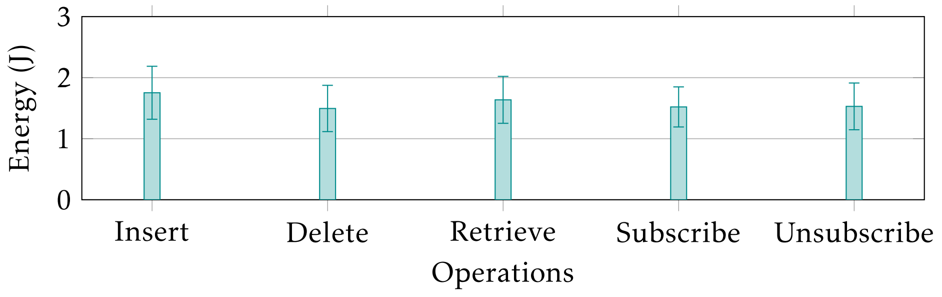


Figure 4.19: Energy usage when issuing an operation in Thyme-DCS Android [46].

instance, issuing an insert operation consumes around 2 Joule. However, it is not easy to understand the energy consumption in real terms for the average user looking into such small measurements.

To make it easier to fully grasp the energy consumption measurements, we made a different experiment. Figure 4.20 displays a breakdown of the energy consumption of a device while executing a specific operation in a closed loop for *one minute*. This would represent a very intense scenario, however it will be useful to test the worst case in energy consumption. Standby mobile phones, i.e., only connected to the Wi-Fi router (without Internet access), consume around 61 Joule. When running the application, the battery consumption increases by 6 Joule to about 67 Joule; an increase caused by the periodic sending of cell management messages to neighboring nodes. The energy consumed by the different operations is mostly equivalent. On average, delete and (un)subscribe are the more energy friendly operations, consuming around 23 Joule, for a total of 90 Joule. On the other end, with the insert operation the total energy consumption is about 103 Joule.

Putting things more into perspective, in the Nexus 9 devices, 1% of battery corresponds to around 960 Joule. According to the presented data, scenarios with *continuous* and *intensive* use of the application for *10 minutes* (publishing photos, receiving notifications and downloading available photos) consume roughly 950 Joules, which represents about 1% of the device's complete battery charge. A value we claim is quite reasonable for such intensive use.

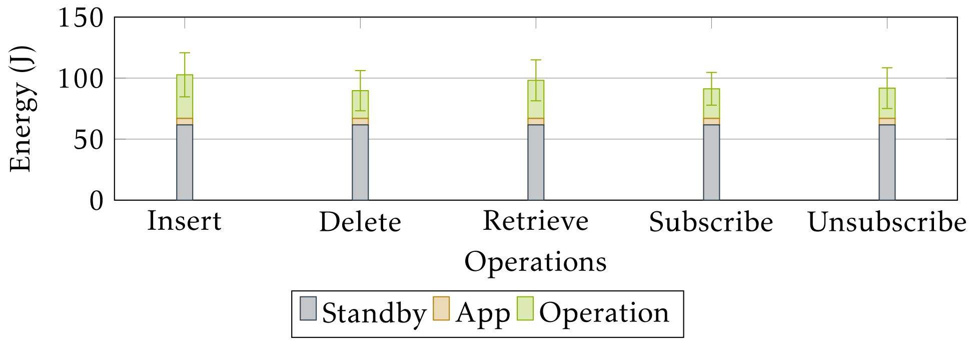


Figure 4.20: Energy usage when issuing an operation in a closed loop during one minute in Thyme-DCS Android [46].

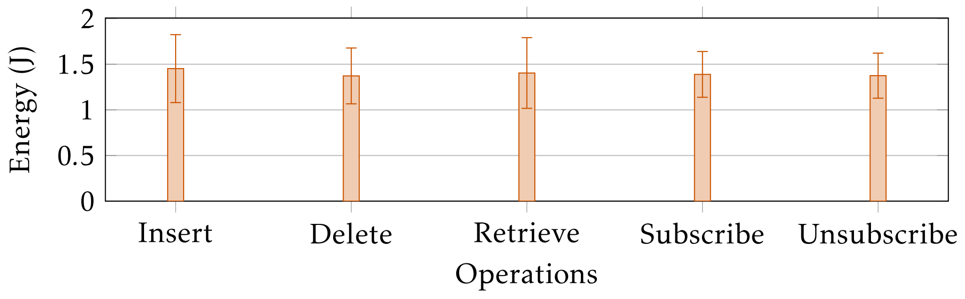


Figure 4.21: Energy usage when processing a request in Thyme-DCS Android [46].

Processing a Request. Since we are talking about a collaborative and distributed system, nodes also work on behalf of each other, i.e., on behalf of the system. Naturally, processing an operation request requires a node to do some computations (and possibly send other messages), and ultimately, spend some battery charge.

Figure 4.21 displays the energy costs involved in processing each type of operation. Looking at this plot, we can observe that energy consumption is similar among all the operations. On average, processing an operation request spends around 1.5 Joule. We argue such a low value is acceptable, even more so since work inside a cell is balanced among all the nodes in the cell. Even in a worst case scenario, by processing requests in a closed loop during 10 minutes, a node will spend, on average, about 830 Joule, which represents a consumption of less than 1% of a Nexus 9 total battery charge.

Cell Maintenance. The Thyme-DCS routing layer is based on a GHT (§4.5.6.1), which is based on the notion of virtual nodes or cells (comprised by physical nodes). Thus, physical nodes inside a cell work on behalf of their cell (e.g., through active replication; §4.5.2.1). As such, the maintenance of a cell requires some computations and an increase in energy consumption of the mobile devices. When a node processes an operation request, all its cell neighbors will have to update their state accordingly.

Figure 4.22 shows the energy consumption of a node during the cell maintenance required for each operation. From this plot, we can verify that the energy consumption for cell maintenance is comparable to the cost of processing the operation request itself (presenting only a negligible decrease). This comes from the fact that when processing the specific request, these nodes do not have to send further messages (i.e., they only process the one they received). Similarly to the rationale followed previously, a node processing the cell maintenance requests continuously during 10 minutes, on average would spend less than 800 Joule, which would mean an expenditure of around 0.8% of the Nexus 9 total battery charge.

Total Energy Cost. Since mobile devices can be simultaneously issuing operations and processing operation requests, and still have to participate in their cells' maintenance, we have to account for these three energy components. Following the same worst case

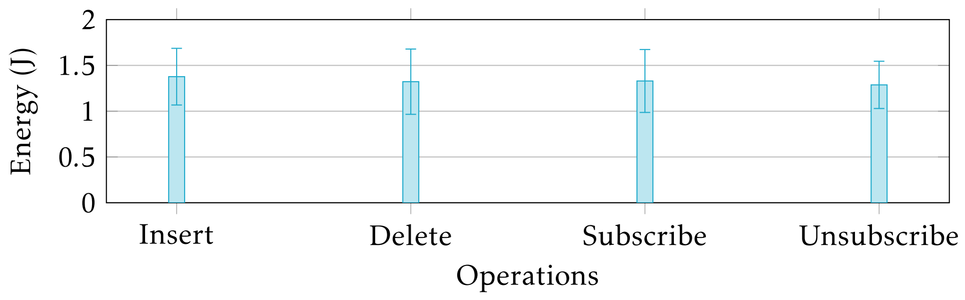


Figure 4.22: Energy usage in the cell maintenance in Thyme-DCS Android [46].

scenario (issuing and processing operations for 10 minutes in a closed loop), and summing up these three components, we get

$$950 + 830 + 800 = 2580 \text{ Joule}$$

corresponding to around 2.7% of the Nexus 9 total battery charge. A value we argue is quite acceptable for such an *intensive* use of the application.

According to these numbers, to deplete 50% of the device's battery charge, a client would need to keep this intensive use continuously for *more than three hours*. In a moderate usage scenario and for a device with a complete battery charge, we estimate a battery life of well over six hours (i.e., a time interval well capable of accommodating a wide range of social events).

4.10 Concluding Remarks

In this chapter, we describe the design of Thyme, a novel storage system implementing TARS for wireless edge environments, enabling applications to be notified as relevant data is generated and stored. We detail two different approaches to Thyme: Thyme-LS follows a lightweight unstructured approach using local storage and query flooding, while Thyme-DCS employs a DCS approach using a storage substrate built over a cell-based GHT for wireless networks. In addition, we describe our implementation of the Thyme-DCS approach as an Android library and its use to develop a proof-of-concept photo sharing application. The innovative characteristics of Thyme offer a novel way for sharing and accessing data that has been previously stored, or is being generated in quasi-real-time, in a network of mobile devices.

The three parts of our evaluation are complementary to each other, showing different facets of our approaches. Overall, the evaluation shows that Thyme allows the notification and retrieval of relevant data with low overhead and latency, even under node failures. However, all the presented approaches display different behaviors and each may be best suited for scenarios with specific characteristics. In general, we show that the developed approaches exhibit a good performance and low energy consumption in the target environments (and under various conditions).

4.10.1 Discussion

As a conclusion, Thyme-ES presents a baseline. It has an external, centralized component where all the system's data is stored. Being a centralized server, it presents itself as a bottleneck and a single point of failure. Thus, this approach assumes it never fails, otherwise the service will become completely unavailable. If that assumption is not an issue, then Thyme-ES can be an option, but only for small scenarios. As our experiments show, increasing the number of nodes in this approach leads to an increase in operation latency and a decrease in operation success ratio resulting from the central component being a bottleneck where all nodes compete for its resources (§4.8.3.1). Note that in our experiments, the centralized component resided close to the client nodes. If that was not the case and it was located in the cloud, we would see much higher latencies.

Due to its flooding approach, Thyme-LS causes far more interferences than Thyme-DCS. This is exacerbated by the number of nodes in the system (§4.8.3.1). Churn is also a problem for Thyme-LS, because insert operations are executed locally and there are no replication mechanisms in place (§4.8.3.2). Thus, if a node fails, all the data it stored will become unavailable, representing some kind of partial failure (because only that node's data becomes unavailable). Also, its flooding approach means that as the number of nodes increases, so does the amount of traffic and interferences. In summary, Thyme-LS is more suitable for smaller scenarios (i.e., with a small number of nodes) with no strong data availability requirements.

In turn, Thyme-DCS leverages geographic routing to employ replication, and location-aware data retrieval (§4.8.3.2). However, one-hop broadcast is unreliable by nature, thus the assumption that every node inside a cell has the same state needs to be relaxed (§4.8.3.3). Nonetheless, its cluster-based GHT deals well with the increase number of nodes in the system and its churn. So, Thyme-DCS is more suitable for larger scenarios with moderate mobility patterns, and data availability requirements (being able to cope with reasonable levels of churn).

Regarding the real world experiments and our proof-of-concept Android application, results show adequate response times for interactive usage and low battery consumption. Yet, the work each node has to do on behalf of the system grows linearly with the amount of work delegated on the cell where they reside. This load can be reduced by using partial replication techniques, for instance, when the cell's population surpasses a given threshold. However, even with the use of full replication in cells, our experiments show that the application can be used during short and medium duration events with no risk of rapidly discharging the devices' battery.

In sum, these three approaches have very different characteristics. Which one is appropriate for a specific setting will depend on the conditions of the environment and the nature of the workload. Thus, we stress that Thyme-DCS is not always the approach of choice, but rather that under some conditions it is preferable. In fact, the perfect case is a system that embodies all of these approaches, and application developers can choose which to use according to the task at hand.

Recalling the logic layed in §3.4, the TARS concept makes a fundamental *overhead shift*. That is, the overhead from the stakeholders that benefit more from this approach—users requesting data—is reduced (compared with the explicit search approach), and is *moved* to the stakeholders that do not benefit directly from it—users that have the data and can provide it. Here, with the replication mechanisms of Thyme-DCS, note that when a node obtains a data object, it becomes a new source for that same object (i.e., a passive replica). Thus, it goes from one side of the stakeholders to the other. Namely, it goes from the side of the users requesting data, to the side of the stakeholders that contribute to the system (like a seeder in a P2P file sharing application).

4.10.2 Future Work

This work can be seen has a first step towards a data storage and dissemination system for a wide-area setting, like a campus or a music festival. In this scenario, data will still be stored in the devices, and communication will mostly be D2D to offload it from the network infrastructure.

There are however several open issues, of which we highlight the following. Non-contiguous spaces, such as the ones composed of multiple Wi-Fi access points: more sophisticated hashing functions and/or maybe the use of cloudlets may allow to cope with such environments. Rapidly state-changing cells: cells may be populated or not, being this state managed by the GHT. However, with high mobility, this state may change rapidly, leading to overheads due to the need of transferring data between devices, and ultimately causing some of this data to be lost, if there is no time to make the necessary backups. A hierarchical cell organization, or a partial replication approach may be possible directions. We also highlight as future work the integration of this approach with opportunistic infrastructure support [242], privacy and security concerns in this type of environments (mainly access control and trust), and tackling the issues raised by handling large data objects.

4.10.3 Publications

The work reported in this chapter resulted in the following publications:

- **It's About Thyme: On the Design and Implementation of a Time-Aware Reactive Storage System for Pervasive Edge Computing Environments [241].** *João A. Silva*, Filipe Cerqueira, Hervé Paulino, João M. Lourenço, João Leitão, Nuno Prequiça. In Elsevier Future Generation Computer Systems (**FGCS**). 2021.

- **Time-Aware Reactive Storage in Wireless Edge Environments [245].** *João A. Silva*, Hervé Paulino, João M. Lourenço, João Leitão, and Nuno Preguiça. In Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (**MobiQuitous**). Houston, Texas, USA, 2019.

- **Time-Aware Publish/Subscribe for Networks of Mobile Devices [244].** *João A. Silva*, Hervé Paulino, João M. Lourenço, João Leitão, Nuno Preguiça. **arXiv**:1801.00297. 2017.
- **Towards a Persistent Publish/Subscribe System for Networks of Mobile Devices [47].** Filipe Cerqueira, *João A. Silva*, João M. Lourenço, Hervé Paulino. In Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets (**MECC@Middleware**). Las Vegas, Nevada, USA, 2017.
- **Um Sistema Publicador/Subscritor com Persistência de Dados para Redes de Dispositivos Móveis [48].** Filipe Cerqueira, *João A. Silva*, João M. Lourenço, Hervé Paulino. In Proceedings of the 9th Simpósio Nacional de Informática (**INForum**). Short paper. Aveiro, Portugal, 2017.

Chapter 5

Parsley: A Resilient DHT with Dynamic Data Sharding

"There is no comparison between what's lost by failing and what's lost for not trying."
— Francis Bacon

This chapter tackles some issues that can be found in the Thyme-DCS approach (§4.5). Namely, the dynamic population of cells (or groups of peers), i.e., the overlay being able to handle the abrupt entry or exit of peers from any cell. Also, the case of data hot-spots, i.e., when some tags are much more popular than others, overloading the cells/groups responsible for managing them. Thus, it reports about managing highly dynamic device population and storage workload imbalances in distributed hash tables (DHTs).

Specifically, in this chapter we describe Parsley, a resilient group-based DHT with a preemptive peer relocation technique and a dynamic data sharding mechanism. This work, also being a flexible DHT, it can be leveraged in any of the levels of the network hierarchy. That is, end-user devices can be the peers of the overlay, or the edge servers, or even machines in the cloud[1].

The first section, §5.1, presents some context and motivation for the addressed issues. Then, in §5.2, we review some related work and compare our approach with it. Next, we detail our resilient group-based DHT solution and its preemptive peer relocation technique in §5.3. In §5.4, we describe our dynamic data sharding mechanism for dealing with storage hot-spots. After, in §5.5, we report our experimental evaluation of the implemented prototype in PeerSim. The chapter is concluded in §5.6, by presenting our main findings and some future work.

[1]It can also be applied to a mix thereof, but this could give rise to other issues that are outside the scope of the work presented in this chapter.

5.1 Introduction

Since their inception, DHTs [211, 221, 256] are an important building block in many distributed systems [30, 45, 68, 72, 94, 144, 145, 272]. They provide a scalable lookup service, used to build complex large-scale systems such as distributed storage [30, 68, 72, 145], instant messaging [272], application-level multicast [45], web caching [94], or file sharing [144]. Peers self-organize into a specific network topology (or *overlay*) to provide key-based routing (KBR), efficiently mapping a given key onto a peer in the overlay, called the *key owner*. All this is done through a single operation: `route(message, key)`. On top of KBR, DHTs usually implement data storage by associating a value (i.e., a data item) with each key, and storing the key-value pair at the key owner [68].

Hashing [133] is used as the key space partitioning scheme, i.e., to assign ownership of a key range to a specific peer, *uniformly* spreading keys among peers, and thus achieving a *balanced* load across the overlay. To work efficiently, DHTs assume: i) keys are *uniformly* accessed, both for query and store operations (i.e., reads and writes); and ii) both the size and amount of values mapping to a key have a *small variance*. However, these assumptions do not occur in several scenarios, e.g., file sharing or content indexing [144, 215, 263]. Some files are more popular than others, and they can have disparate sizes. Also, when indexing through keywords, some are going to be more popular than others, resulting in some keys mapping to much more values than others. Hence, these scenarios present many sources of *load imbalance*.

These query and storage asymmetries originate hot-spots that arise due to the usually *non-uniform* distribution of data (often power laws) [297]. *Query hot-spots*, i.e., a few keys being queried very often, can cause network bottlenecks on the peers storing those popular keys. Other problems emerge when some ranges of the key space are more popular than others, overloading just some parts of the overlay. In turn, *storage hot-spots* appear when large or many different values (of possibly skewed size) are mapped to a *single* DHT key, overloading the owners of those keys. Ultimately, these hot-spots and consequent bottlenecks hamper the scalability of DHTs. Moreover, in line with the soaring advance of edge computing [96, 240], these load imbalance issues are even more exacerbated when referring to edge devices, like micro-computers, cloudlets or even mobile devices, known for being resource-constrained, mostly regarding storage space or communications (and even energy).

In this chapter, we address these problems of load imbalance in DHTs through the combination of unstructured and structured overlay techniques. We present Parsley, a novel DHT that provides robust and efficient data storage while enabling load balancing both for query and storage hot-spots. The unstructured component comes from peers being clustered into *groups* of flexible size, working collaboratively within them. In turn, each group acts as a *virtual peer* in the structured layer.

Groups fulfill a two-fold goal: they are used to simplify data replication management, and enable query load balancing. This is achieved through data replication inside each group, allowing to spread the query load among the group peers, while at the same time providing increased resilience to churn. Groups split and merge throughout the overlay lifetime, according to criteria based on group size and amount of stored data, also adapting to the (possibly non-uniform) data distribution in the key space. We diverge from previous work in the way we proactively (and reactively) relocate individual peers from larger into smaller groups. We use *peer relocation* between groups as a preemptive measure to avoid merging them (since it requires costly *state transfers*). In sum, all our decisions are carefully crafted in order to impose the minimum disruption possible on the overlay topology.

To balance storage hot-spots, we developed a lightweight *dynamic data sharding* mechanism for popular keys. Here, we get inspiration from the multi-publication replication technique (§2.4.2.3), and apply a variation of it to disjoint partitions of the (multiple) values mapping to a single DHT key. When a group finds that a key is storing too many values (according to some configurable criteria), it starts sharding the key, dynamically partitioning the mapped values among other groups.

We implemented a prototype of Parsley in PeerSim [180] and use it to experimentally validate its performance, detailing and characterizing its behavior in different scenarios. The experimental results reveal that our approach has the following benefits with regard to previous work: i) the preemptive peer relocation mechanism reduces the amount of required group merges throughout the overlay lifetime, consequently decreasing the bandwidth needed for state transfer between groups; and ii) the dynamic data sharding mechanism allows to more evenly spread the (storage) load imposed by popular keys among several groups, helping reduce storage bottlenecks.

In summary, the main contributions of the work presented in this chapter are the following: 1) a flexible and resilient group-based DHT with a preemptive peer relocation (push-pull) mechanism; 2) a lightweight dynamic data sharding mechanism; and 3) a comprehensive experimental evaluation of our prototype in PeerSim.

5.2 Related Work

As already mentioned, one of DHTs' fundamental issues is that peers or keys may not be uniformly distributed in the key space [144, 215, 263]. Thus, some peers may be overloaded, having to store many keys or answer many queries, while others may be relatively idle. Regarding techniques to address the issues with query hot-spots, we have already described a comprehensive amount in §2.4.2.2.

The works more closely related with our proposed solution are Rollerchain [192], MobiStore [139], and DEB Tree [158]. We present their main differences in Table 5.1, where n is the total number of groups in the system.

Rollerchain is a group-based DHT, focused on efficient replication. Groups are split or merged according to their size and load, and only merges require state transfer between groups. Nonetheless, peers that are group leaders have many responsibilities (e.g.,

Table 5.1: Comparison of Parsley with other related proposals.

	Routing	Replication	Peer Relocation	Sharding
Rollerchain	O($log\ n$)	Group-based	No	No
MobiStore	O(1)	Group-based	Push	No
DEB Tree	—	—	—	Tree
Parsley	O($log\ n$)	Group-based	Push-Pull	Flat

an intricate virtual link management), possibly entailing more work than other group peers. It does not employ any kind of peer relocation mechanism, thus requiring group merges when a simple peer transfer could suffice to prevent a merge. Also, the authors completely disregard the problem raised by peers joining the overlay concurrently with groups splitting or merging, severely hampering the freedom of peers to enter the overlay. Its group-based approach with replication addresses the issues with query hot-spots. However, it does not address the problems caused by storage hot-spots.

MobiStore is a one-hop group-based DHT, targeting mobile environments. Being a one-hop DHT, implies storing more state (in this case, amounting to four different routing tables storing various information), entailing overheads both in terms of storage and communication. Possibly also incurring in other overheads to manage and keep that state up-to-date. Contrary to Parsley, due to its group management mechanism, splits and merges require communication with *all* other groups. Additionally, group merges can only happen one at a time in the entire overlay, and, in some cases, splits may require state transfer between groups. MobiStore also employs a peer relocation mechanism. However, it only allows a *few* peers to move in each group (the called loose peers) through a push strategy. Only a single peer can be transferred at a time system-wide (almost as a "stop the world" policy), also requiring communication with *all* the other groups in the overlay. Furthermore, this peer relocation mechanism is fundamentally different from the one in Parsley—it is only used to address the load balancing problem of answering requests. Thus, it is not used to try to prevent topology changes (and with that state transfer between groups). It also does not address the problem of storage hot-spots.

DEB Tree provides a solution for storage hot-spots in DHT-based inverted indexes. It uses a B^{+}-tree structure over a *generic* DHT that adapts dynamically to the object size, ensuring a uniform storage distribution despite the object size variation. When inserting data in the DHT, it converts a very large object into multiple bounded size blocks. A balanced tree-based approach is used to split the index associated with a popular key across multiple peers. Thus, each tree instance stores the objects for a particular key(word), meaning each index key(word) will have an unique DEB Tree, encompassing some storage and management overheads. The tree structure is composed by a root node and child nodes, where the last level of nodes are called leaf nodes. In the tree structure, leaf nodes store objects, and are all at the same tree level. In turn, internal nodes serve only for locating leaf nodes and only contain child node keys (i.e., they do not contain any data). The number of elements in any node is bounded by the tree's degree: for internal nodes, the degree influences the number of child node keys it contains; and for leaf nodes, it influences the number of objects the node stores. Despite tackling storage hot-spots, this approach requires DHT routing for each step of the tree traversal, imposing high latency, and requiring more bandwidth for each DHT operation. To alleviate this issue, they employ caching of tree internal nodes, which works best in more stable scenarios. Also, to be tolerant to concurrency issues regarding the structural integrity of the tree and of the stored data, some operations may be delayed or retried. It provides no fault-tolerance regarding peer failures and churn, meaning that tree internal nodes can be recovered, but that is not the case with leaf nodes, which can lead to data loss. Further, the authors completely disregard the intricacies and complexity of tree rebalancing in this distributed setting, which can entail multiple changes in different parts of the tree, requiring a considerable amount of communication.

In turn, Parsley's flexibility allows the configuration of several of its procedures' decision criteria. Splits never require state transfer between groups, and the DHT logic is carefully crafted so that group leaders do not have to execute much more work than other peers. Also, its preemptive peer relocation mechanism employs a push-pull strategy, allowing any peer to move, and only requiring communication between the involved groups. Lastly, to tackle storage hot-spots, Parsley employs a sharding mechanism on a flat structure, inspired by the multi-publication replication technique, thus entailing a small communication overhead.

5.3 Chopping Parsley: A Resilient DHT

Parsley is a group-based DHT that provides robust and efficient data storage while enabling load balancing both for query and storage hot-spots. It dynamically manages groups of peers by combining unstructured and structured overlay techniques. This group-based approach mainly serves the dual purpose of enabling query load distribution and fault tolerance.

Usually, DHTs associate a single value with a key. However, to be more flexible and allow indexing, in Parsley, keys map to a *set* of values, distinguished through unique object identifiers (oids). Also, it provides the three typical DHT operations: `put(K,V)`, `get(K)`, and `remove(K)`.

5.3.1 System Model

We consider a classical asynchronous model comprised of several processes running the same protocol. We assume a message passing environment in which all processes communicate with each other by sending messages over a communication channel. We also consider the classical crash-stop failure model: processes can fail, i.e., halt prematurely,

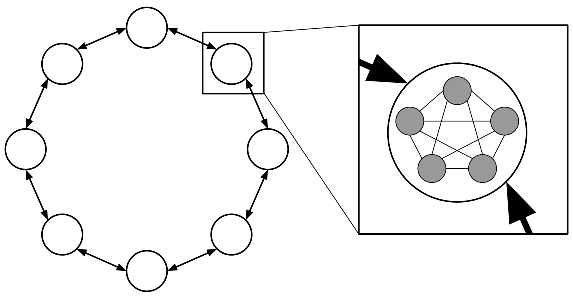


Figure 5.1: Parsley's architecture overview.

but do not behave maliciously. Each process has a globally unique identifier, and we use the term *peer* to represent a process that is running on a particular machine.

All communication resorts to TCP connections. The use of TCP is relevant because it allows the communication between peers to be network-friendly, as we leverage in TCP flow control mechanisms, and also makes it possible to model the system without considering message losses between peers. Additionally, TCP is used as an unreliable failure detector [149, 192]. The failure detector is used, for instance, to expedite the detection of failed peers, allowing the protocol to take the adequate actions to ensure the correctness of the overlay.

5.3.2 Overview and Definitions

Parsley's unstructured component clusters peers into groups. A group is a fully connected cluster of peers (i.e., a clique), with a flexible size within a defined interval, where peers work collaboratively to function as a single (logical) virtual peer in the structured component. In turn, the structured component is based on a typical ring DHT, such as Chord [256], and uses the same mechanisms (§2.4.2.1), i.e., pointers, stabilization, finger tables, and successor lists. Thus, the structured component (i.e., the DHT) is a doubly linked ring composed of groups provided by the unstructured component. Fig. 5.1 depicts a simple representation of Parsley.

To simplify coordination inside a group in some procedures, each group has a *group leader*, i.e., a peer responsible for (minimally) coordinating some of the most delicate moments of a group's life. There are many ways of electing a leader. In our case, we use an "eventual" election mechanism: the member with the lowest identifier in each group acts as its leader. Occasionally, more than one peer may see themselves as the group leader. However, this does not affect correctness, as the leader is only used to reduce the protocol signaling costs. Also, if no peer sees itself as the leader, this only delays the protocol's progress until the group maintenance procedure corrects that, enabling one of the members to see itself as the leader (see the group maintenance procedure in §5.3.3.3).

In Parsley, groups have a configurable minimum and maximum size, l and h, respectively, where $h \geq (l \times 2) - 1$. The group size allows to fine tune the trade-off between group maintenance (i.e., replica monitoring and storage load), and tolerance to churn and

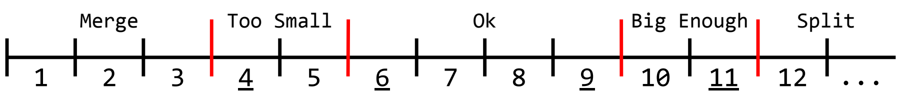


Figure 5.2: Group size with hard limits set to 4 and 11, and soft limits set to 6 and 9.

query load balancing. These are *hard limits*, thus exceeding them means having to take effective actions, i.e., a topology change. Accordingly, a group that exceeds the maximum allowed size (i.e., $|group| > h$) needs to split into two (see the split procedure in §5.3.3.4). On the contrary, a group that reaches below the minimum size (i.e., $|group| < l$) needs to merge with another one (see the merge procedure in §5.3.3.5).

However, contrasting with previous work, we also employ *soft limits*, allowing us to take some preemptive actions before reaching the hard limits (e.g., coercive merge in §5.3.3.5, and preemptive peer relocation in §5.3.3.6). These soft limits also have a configurable minimum and maximum, l' and h', respectively, creating a desired target interval for group size, where $l \leq l' < h' \leq h$. Now, as represented in Figure 5.2, we have additional intervals where groups with their size in the range $]h', h]$ are considered to be *big enough*, and groups with their size in the range $[l, l'[$ are considered to be *too small*. Thus, a group tries to maintain a target size in the range $[l', h']$. With these soft limits, preferably, the following condition should be met: $h \geq (l' \times 2) - 1$. As an example, Figure 5.2 defines the following limits: $l = 4$, $l' = 6$, $h' = 9$, and $h = 11$. The group size is checked periodically by the group leader (if the group is not in a cool down period, and is not splitting nor merging). By doing this periodically, we impose an implicit maximum rate at which groups can split or merge, and consequently the overlay topology can change (like a throttling mechanism).

Since we are mostly storage-oriented, we define a *popular key* as one having many values mapped to it, i.e., the more values a key has the more popular it is. In our case, a key having more values than a configurable popularity threshold is considered popular (see the hot-spot detectors in §5.4.2). In the same line, we define the *load of a group* as the number of keys it stores, normalized to the number of peers in the group, i.e., $\frac{|keys|}{|group|}$. Thus, a group is overloaded if this ratio is over a configurable threshold. This notion of load is only used to decide whether to split groups or not. Since group splits are only able to divide whole keys, this notion suffices.

5.3.3 A Group-Based DHT

Peers inside a group work collaboratively, replicating among them the necessary information to maintain the DHT topology and the data stored by the group. This replication not only increases data resilience, but also allows peers to share the load of answering queries for data they store. Here, the group-based approach together with its split/merge logic allows Parsley to tackle both the issues of query hot-spots and specific key ranges being overloaded.

In Parsley, the unstructured component manages the groups of peers, while the structured component executes the DHT maintenance protocols.

In the pseudo-code presented in the next sections, we use an object-oriented notation, where *a.bar* and *a.*foo() represent, respectively, accessing attribute *bar* and function foo from object *a*.

5.3.3.1 Ring Maintenance

Parsley's structured component is inspired in Chord, as such it uses the same ring maintenance procedure (Algorithm 2.1 in §2.4.2.1). However, here, the maintenance procedure is executed at group level. When triggered, a peer contacts another random peer from the intended group.

In the maintenance of both successor pointers (i.e., stabilization) and finger table entries (i.e., fix finger), we employ an adaptive periodic timer. Thus, as long as the result from the maintenance procedure is as expected, the periodic interval is incremented (by a predefined increment value), until reaching a configurable maximum. Whenever the result is different from what was expected, the interval goes back to the configured minimum, and this logic repeats. In the end, this mechanism tries to save bandwidth, by reducing communication in stable moments of the overlay. When something unexpected or different is detected, to be conservative, we revert to a smaller maintenance interval.

5.3.3.2 Joining a Group

To enter the system, a peer sends a join request to another one already in the overlay (i.e., its gateway peer), starting a random walk that probes for suitable groups to join. The natural exception is the first peer to enter (in this case, create) the overlay; it only has to create the seed group with a random group identifier (gid). The random walk terminates when a group accepts the joining peer following a programmable *new peer acceptance logic*, or when the message time-to-live (TTL) expires (and the current group is forced to accept the new peer). After being accepted, the peer receives a join reply, and uses that information to update its internal state and to establish connections to its new group members. This reply also carries the current state of the group's key-value store.

Since one of our main goals is to balance load among the overlay groups, our current implementation of the *new peer acceptance logic* (that we provide as a default implementation, shown in Algorithm 5.1) is divided in two parts. First, a new peer is accepted if the current group is overloaded, in order to increase group-wide load balancing, even if it means splitting the group afterwards (line 4). Next, if the current group is too small, the new peer is also accepted in order to increase the group's reliability (line 6). In case the current group is big enough (and is also not overloaded), the new peer is rejected and the random walk continues (line 8). Otherwise, if none of the previous conditions are triggered, we move to the second part (lines 11–14). Here, the probability of rejecting this new peer is inversely proportional to the number of hops already traveled, and it

Algorithm 5.1 New peer acceptance logic default implementation in Parsley.

```
1: $group \leftarrow [id : \perp, view : \emptyset]$                          ▷ my current group
2: $store \leftarrow \emptyset$                                            ▷ key-value store $[k \mapsto \emptyset]$

3: function SHOULDACCEPTPEER($p, visited, hops$)
4:     if $\frac{|\text{KEYS}(store)|}{|group.view|}$ > MAX_LOAD_THRESHOLD then   ▷ is my group overloaded?
5:         return $true$
6:     else if $|group.view| < l'$ then                                 ▷ is my group too small?
7:         return $true$
8:     else if $|group.view| > h'$ then                                 ▷ is my group big enough?
9:         return $false$
10:    else                                                          ▷ external decision criteria
11:        if RAND( ) < 1.0 − ($hops$/JOIN_TTL) then
12:            $s \leftarrow$ GETSMALLESTKNOWNGROUP($visited$)          ▷ excluding already visited groups
13:            if $s \neq \perp \wedge |s.view| < |group.view|$ then
14:                return $false$
15:    return $true$
```

is only rejected if the current group knows any other group smaller than itself (through function getSmallestKnownGroup, which excludes groups already visited by the message random walk). If that is the case, the random walk continues (and it most likely will be forwarded to one of those smaller groups). In the end, this logic causes heavily loaded (and smaller) groups to attract new peers in order to share their load.

The programmer-defined *new peer acceptance logic* has access to a plethora of system information, namely: the joining peer id, the join message visited groups and number of hops, the current peer's pointers (predecessor, successor, successor list, and finger table) and local store and shards, and the current peer's group and load.

5.3.3.3 Group Maintenance

As peers can enter and leave the overlay freely, and can even crash, the group membership maintained by different peers in the same group may diverge. In order to increase the intra-group consistency, a simple gossip-based anti-entropy procedure is executed inside each group (which is also very low cost).

Periodically, with a given probability γ (which can be very small [150]), every peer selects another one in the group and sends it a maintenance message containing information about the group: 1) its membership; 2) the ring pointers (i.e., predecessor and successor); and 3) the hash of the key-value store. This allows to detect missing peers in the membership, and missing or conflicting ring pointers. If the receiving peer detects some missing peer or pointer in the received information, it replies back with a similar message to the sender.

The key-value store hash enables a lazy data replication (and synchronization) scheme. If a peer detects that its hash differs from that of a neighbor, it triggers a pair-wise exchange, where they swap their full list of keys, so that both can request their missing key-value pairs from each other.

Also, to avoid possible inconsistencies, received group maintenance messages are not processed by peers that: i) are in the cool down period; ii) are splitting or merging; iii) are relocating between groups; or iv) are not from the same group (this can happen due to peer relocations; and in this case, the receiving peer sends a disconnect message to try to fix the wrong membership of the sending peer).

5.3.3.4 Splitting Groups

The split procedure divides a group into two new groups having roughly the same size and load. This reduces the cost of replicating data among the group peers, and at the same time the amount of data each peer has to store. A group split happens in two situations:

1. when the group size is above the configured maximum threshold; or
2. when, although the group size did not reach the maximum threshold, the group is overloaded and is big enough[2].

Because of situation 2, the group size limits should ideally follow: $h' \geq (l' \times 2) - 1$. Making that, by splitting a group in that situation it will not create any group smaller that l' (making it too small). For instance, the example in Figure 5.2 does not meet this criterion. With $h' = 9$, a group can be split when its size is ten, creating two groups that are too small for the configured parameters.

The split procedure is initiated by the group leader when, after the periodic group size check, detects that one of the split criteria is met. Then, the group leader creates two new groups, divides the peers randomly between the two, and sends them a message with that mapping.

To avoid inducing artificial churn, one of the newly generated groups keeps the identifier of the original group and the other becomes its *predecessor*, by assuming an identifier that allows it to become the owner of *half* of the original group's key-value store. Thus, a group split does not require any kind of state transfer among the peers, having only to locally discard keys. As a result of this strategy, if keys are not uniformly distributed in the key space, group identifiers will dynamically adapt to follow a similar distribution and consequently will also not be uniformly distributed (which can have impact on routing).

When choosing the identifier of the new group, Parsley also takes into account the *size* of the values mapping to each key, i.e., we find the identifier that best divides the total *bytes* of the local key-value store in half. In the special case when the number of keys is less than two, it is impossible to divide keys even further, thus we divide the key range in half (i.e., between my predecessor and myself). In Figure 5.3, a split can be seen as going from Figure 5.3a to Figure 5.3b, where group 12 triggered a split, originating group 9 (a predecessor of itself).

In Figure 5.4, with all keys having the same number of values and roughly the same size (thus, being perfectly balanced), we can map to the previous example, where the newly created group 9 keeps keys 6 and 9, while group 12 keeps keys 11 and 12. In turn,

[2]This is only allowed if the split does not cause the resulting groups' sizes to fall below the minimum group size limit l (which can happen for some configurations of the group size parameters).

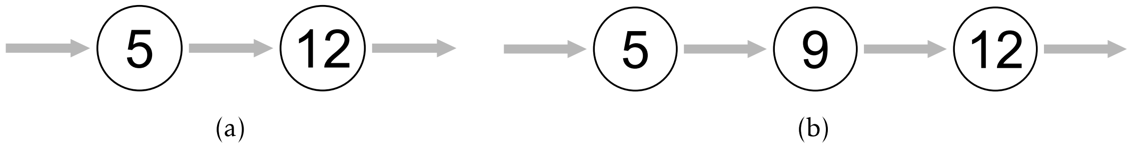


Figure 5.3: Group split example scenario.

in a scenario where values' distribution is imbalanced, the keys mapping resulting from a group split can be quite different. For instance, with all values still having the same size, imagine that key 6 has now 10 values, while the other keys have only one value. When group 12 splits, having a total of 13 values, Parsley will try to divide these as evenly as possible (in this case, the perfect balance is each resulting group storing 6.5 values). Thus, since a split is only able to "break" whole keys, the new group will be assigned identifier 7, causing it to only store key 6 (with 10 values), and group 12 continues to store keys 9, 11, and 12 (amounting to 3 values).

The split procedure mechanism also handles the failure of participating peers during a split. There is no problem if a regular peer fails. The others will notice that through the unreliable failure detector, and will remove it from their local group membership. Since the split is a totally distributed procedure, there is also no problem if the split leader fails. It just happens that no update successor message is sent to the previous predecessor group (to speed up pointer convergence).

Peers that join a group during a split procedure (since they do not known about the split in course) will remain in the original group after the division. In order to speed up the convergence of their pointers and group membership, the gateway peers will send a split fix message to freshly joined peers. In case a gateway peer disappears meanwhile, the state of its joined peers will take some time to become consistent again: i) they will need to remove some peers from their local group membership (i.e., the ones that went to the newly created predecessor group during the split); and ii) they will need to update their predecessor pointers (by receiving a notify message).

5.3.3.5 Merging Groups

A group merges with another when its leader detects that its group size is below the predefined minimum threshold. This procedure works to prevent data from being lost by combining a group with its *successor*. By merging with its successor, we ensure that peers belonging to the two merging groups retain their key-value pairs. Looking at Figure 5.3, a merge can be seen as going in the opposite direction, from Figure 5.3b to Figure 5.3a,

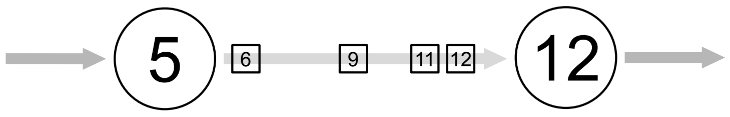
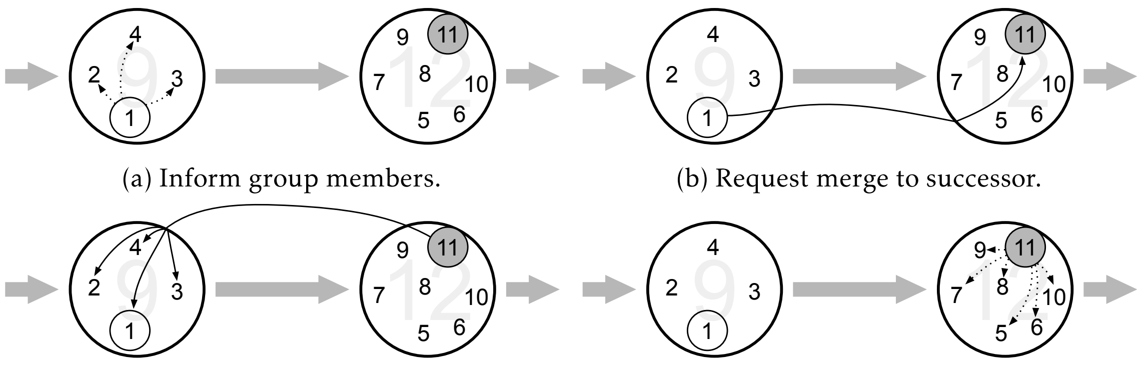


Figure 5.4: Key range in a group. The squares are keys, all owned by group 12.



(a) Inform group members.

(b) Request merge to successor.

(c) Reply to merging peers.

(d) Update successor members.

Figure 5.5: Example scenario of the group merge communication workflow.

where group 9 triggered a merge, fusing with group 12 (its successor).

However, a state transfer will always be required, since peers need to exchange their key-value pairs with peers of the neighbor group. Although this procedure increases the amount of data at the successor group, the merging group's peers also join in this effort, mitigating the increase in load.

Figure 5.5 depicts an example scenario of the group merge phases. Similarly to the split procedure, a group merge is started by the leader of the merging group (node 1 in the figure), by informing its peers (Figure 5.5a), and then requesting a merge to its successor (Figure 5.5b). Then, the successor group replies to the entire merging group (through the peer that received the merge request—the merge coordinator; node 11 in the figure), with the information regarding their new pointers (Figure 5.5c). When receiving that merge reply, peers apply the corresponding information to their state and open new connections to their new neighbor peers. Concurrently with this phase, the merge coordinator also informs its group members of this merge, so that they are also able to update their state (Figure 5.5d).

Regarding keys exchange after a merge, we take a conservative approach. For instance, Rollerchain does not exchange keys after a merge, and lets the group maintenance procedure handle that. However, depending on the amount of churn taking place, a strategy like Rollerchain's might not be enough and lead to the loss of data (e.g., in high churn scenarios). Since the group maintenance procedure is periodic (and probabilistic), it might be too slow. Additionally, since it is pair-wise (i.e., one maintenance message involves only two peers), it might leave crucial peers/replicas out, which can disappear before the appropriate backups are executed. Thus, contrary to related work, after a merge, Parsley explicitly exchanges keys between the two merging groups. Here, peers use the same mechanism as in the group maintenance, except that they do not swap their full list of keys before, and exchange their keys right away. Note that the cost of this state transfer between the two groups is always going to be paid, either eagerly (as Parsley) or lazily (as Rollerchain).

When a group leader detects that a merge should happen, if its predecessor group also needs to merge, it does not go forward with the merge and waits for its predecessor to merge with its group. If this happens, probably this merge is no longer required. However, during moments of churn, pointers might be inconsistent. Thus, if a group leader hits this case a configurable maximum number of times, it suspects its predecessor (i.e., removes the entire pointer) and carries on with the merge.

Peers that join a group during a merge procedure are not informed about the ongoing merge. Then, when the merge ends, the gateway peers of the joining peers will forward to them the result of the merge: if they were in the merging group, they receive a merge reply message; if the were in the successor/merger group, they receive a merge update message. Nonetheless, there is a special case: a joining peer that would become the new group leader (i.e., has the lowest peer id of all the group peers) might try to execute a merge procedure at the same time as the other (ongoing) merge. Thus, to avoid duplicate merges in a group, if a peer joins during a group merge and it is the new group leader, it is informed by the gateway peer about the ongoing merge. Then, the joining peer is only able to trigger a new merge after receiving the previous merge result.

In order for DHTs to scale with the network size, peers only know a small subset of other peers in the overlay. Thus, in Parsley, as in other DHTs, groups are only aware of a (small) part of the overlay, i.e., no group knows all the others. In cases where the overlay looses a large amount of peers (e.g., due to crashes), through our peer relocation mechanism (see §5.3.3.6), big enough groups will push out some of their peers into too small groups, and too small groups will try to ask for peers from big enough groups. However, after some time, groups that were big enough, are no more, and too small groups will no longer know big enough groups to ask for peers from. Nonetheless, there will exist some too small groups in the overlay. Thus, in order to favor group reliability, we devise a *coercive merge* mechanism, where too small groups are coerced to do a merge even when the usual merge criteria is not met. According to this mechanism, a merge is triggered when, during a configurable amount of time, a group: i) is considered to be too small; ii) does not need to split or merge; and iii) is unable to pull peers.

5.3.3.6 Preemptive Peer Relocation

A group is required to merge when its size reaches below the predefined minimum limit, and this calls for a bulky state transfer, as the two merging groups need to exchange their key-value pairs. To avoid this procedure as much as possible and save bandwidth, we devise a preemptive peer relocation (PPR) mechanism, where we proactively and reactively relocate peers from larger into smaller groups. By relocating only a few peers, we reduce both the amount of merge procedures required during the overlay lifetime, and reduce the bandwidth needed to exchange the key-value pairs of the merging groups. In this case, we trade the bulky state transfers required by the merge, for a small state transfer to make the relocating peers up-to-date with their new group. In some sense, we reduce the granularity of the merge, i.e., instead of moving entire groups, alternatively we only move individual peers.

This mechanism uses a push-pull strategy, triggered periodically by the group leader. Groups that are *big enough* volunteer to offer some of their members to groups that are too small. In turn, groups that are *too small* ask for help from groups that are big enough, to see if they can forgo any of their peers. For this mechanism to have some effect, naturally the following condition must hold: $l < l' < h' < h$.

To help amortize the state transfer costs and stabilize the system, peers can only relocate from time to time. After being relocated to a new group, a peer needs to wait a configurable amount of (cool down) time to be able to relocate again—what we call the peer relocation cool down period.

Note that, in order for this mechanism to be lightweight, peers make relocation decisions based on their local (partial, and possibly outdated) information regarding the known groups. Hence, it may be the case that a peer thinks another one is in group A, but it has relocated in the meantime to group Z. Nonetheless, the impact of this issue is kept small through the relocation requests that are sent before the actual relocation. Relocation requests are only accepted by groups that consider themselves to be *too small*, in an attempt to make this mechanism more fruitful.

This mechanism is fully configurable. That is, it can be completely turned off or on. Also, we can toggle individually each of the requests, push or pull.

Pushing Peers. If a group is big enough, i.e., its size is in the range $]h', h]$, it tries to ask some of its group members to go to other (too small) groups in need of peers. For this, the group leader sends a *push request* to some of its neighbor peers, requesting them to relocate to the detected groups in need. Here, the number of desired peers (i.e., to push out) is up to $|group| - h'$.

Pulling Peers. On the other hand, if a group is too small, i.e., its size is in the range $[l, l'[$, it tries to ask peers from other groups to come to its own group. For this, the group leader checks all its known groups (e.g., predecessor, successors, fingers) for big enough ones, and sends a *pull request* to some peers of those groups. Here, the number of desired peers (i.e., to pull in) is up to $h' - |group|$.

Relocation Request. Instead of blindly accepting a relocation request, first, peers send a relocation request to the group where they are supposed to relocate. A peer receiving a relocation request, only accepts it if: i) it is not relocating between groups; and ii) its group is *too small*. If these criteria are met, the peer replies back with a join reply message. Otherwise, it replies with a relocation denied message, and the relocation has no effect. We employ this verification mechanism to avoid allowing unnecessary relocations (that have a direct implication in communication costs). When a relocating peer receives a join reply, it first leaves its old group, and then accepts the received new group.

Avoiding Merge-Caused Splits. To avoid as much as possible that groups have to merge, we leverage on the peer relocation mechanism and employ a simple optimization during the merge procedure. When the successor group receives a merge request, if it detects that the current merge would result in a split afterwards, i.e., $|group| + |successor| > h$, it cancels the current merge and uses the relocation mechanism to *force* some of its group members to go to the merging group.

5.3.3.7 Increasing Fault Tolerance

To increase the robustness of Parsley, we employ some additional mechanisms, namely a passive (partial) view of the overlay, and a recover successor mechanism.

Passive View. Contrasting with related work, we use a mechanism similar to the one described in [149], where each peer keeps a random, unbiased, partial view of the overlay. We call it *passive view*, and is maintained using a low cost background protocol (similar to the group maintenance in §5.3.3.3), based on the exchange of shuffle messages. Whenever a peer removes a correct peer from any of its pointers, or when it receives a request sent by a peer which is not in those pointers, that peer can be added to the passive view. Ultimately, this view is used as a backup during the join random walk, and also when finding or recovering a successor.

Recover Successor. In ring DHTs, a correctness criterion is to maintain a correct successor, thus they keep a list of its nearest successors in the ring that is used to replace failed successors. However, in high churn scenarios, this list might not be enough. Thus, Parsley makes use of another mechanism to help in these cases. When a peer's successor list becomes empty, a recover successor procedure is triggered. This mechanism is similar to the find successor procedure (§2.4.2.1), but on the contrary, it follows the closest succeeding fingers known in each hop. This message has a TTL, and a reply is sent to the requester if the correct interval was found, or if the TTL ends. This means that the received pointer might not be the correct one, but is on the path to such one. Then, the peer will update its successor pointer through the stabilization procedure, until reaching the correct one. Depending on how good was the answer, this might take some time to correct the pointer. Note that this is a probabilistic or best-effort mechanism.

5.3.3.8 Replication & DHT Routing

The replication of key-value pairs among group members is performed using a combination of eager and lazy data replication schemes. First, eager replication is used when a key-value pair is inserted in the DHT, where the peer receiving the request multicasts that same request to all its group members, replicating the key-value pair. Afterwards, the group maintenance procedure executed among group peers works using a lazy replication scheme to maintain replicas (§5.3.3.3).

Note that, since there is no primary copy of the data (as in [99, 192]), we do not provide strong consistency among replicas. However, the maintenance procedure running in each group guarantees that eventually all peers will locally store the group's key-value pairs.

Regarding DHT routing, as Rollerchain [192], we follow Chord's routing approach (described in §2.4.2.1) with some tweaks. The lookups done during DHT routing follow the virtual peers (i.e., groups) maintained in the finger tables. Thus, when a lookup reaches a peer, it uses its finger table to choose a group to be the next hop for the lookup. When forwarding the lookup, it chooses a random peer from the next hop group, thus achieving load balancing by design. As in Chord, the lookup ends when it reaches the predecessor of the target key, which returns its successor as the key owner.

As already described, group identifiers are selected as to promote load balancing of the amount of data stored by each group. As such, group identifiers may not be uniformly distributed in the key space. Since that is not the case with typical DHTs (i.e., they assume peer identifiers are uniformly distributed), their routing mechanisms may not work as expected if applied directly. Thus, the finger table update procedure of Parsley is based in the technique presented by Chord# [235]. Peers periodically update their finger tables. Also, as in Chord, each row of a peer's finger table represents a (virtual) peer at an exponentially increasing distance in the ring. However, this scheme does not rely on the group identifiers. Instead, row i represents a group which is b^i hops away in the ring (where b is a parameter)[3].

The group for each row in the finger table is found by recursion in other group's finger tables. Each row i is assigned using information from the finger table of the group in row $i-1$. For the recursion base, as in Chord, row 0 is populated with the (immediate) successor group. The intuition behind this technique is that row i of group A represents group B, which is located n hops away, then row $i+1$ represents a group $n \times b$ hops away, which corresponds to the group at row i of group B's finger table. Thus, the entry for row $i+1$ of group A is assigned with the group in row i of group B's finger table.

This requires a *constant* amount of messages to update an entry of a group's finger table, whereas Chord's update procedure requires $log(n)$ messages for each entry.

A small handicap of this mechanism is that, since it requires the previous entry in the finger table for an update, if that (previous) entry does not exist, we cannot update the current entry. Additionally, the following entries in the finger table will also not be able to be properly updated. Nevertheless, in practice, the successor lookup works well, even if the finger table is not precisely updated [256].

5.4 Dynamic Data Sharding

Storage hot-spots appear when large or many different values (of possibly skewed size) are mapped to a single DHT key, overloading the owners of those keys. This issue can

[3]In our case, we used $b = 2$, and finger tables of size $log(n)$, being n the number of groups in the overlay.

be easily understood through an example usage of the THYME-DCS approach. For instance, imagine the shared photo gallery application (§4.6.4) being used in a football stadium. When a player scores a goal, several application users all use the app to share photos of this memorable moment with tag “goal”. Suddenly, the nodes of the key owner group/cell are flooded with all those insert operations, and now have to store and manage all the resulting metadata. To tackle this problem, we devise a lightweight dynamic data sharding mechanism for popular keys, that leverages on PARSLEY’s characteristics to effectively distribute the storage load among several groups.

In its essence, this mechanism has some resemblance to the multi-publication replication technique (§2.4.2.3), but here is applied to disjoint partitions of the (multiple) values mapping to a single (popular) key. Thus, allowing to more evenly distribute the storage load of those keys among several groups. When a group finds that a key has too many values (according to a configurable criteria; see §5.4.2), it starts sharding the key, dynamically partitioning the mapped values among other groups.

Promptly, we envision two main directions to achieve this:

1. use a primary hash function h_p applied to the object identifier, $h_p(oid) = i$, to decide which secondary hash function h_s^i to use; or

2. decide randomly which secondary hash function h_s^i to use (i.e., $i = rand()$).

And then, apply the secondary hash function $h_s^i(k)$, being k the key to which the value maps. Option 1 enables a direct `get` operation to a specific value, because every oid determines the value’s shard. However, this option requires *rehashing* every time a key is (re)sharded (which is a very costly operation), because there is a change in the number of shards. In turn, option 2 does not require any kind of rehashing, but a `get` operation to a specific value requires a lookup in *every* shard.

Since we aim for PARSLEY to be flexible, and allow scenarios like (content) indexing, where usually *all* the values mapping to a key are retrieved, we choose the trade-off given by option 2. Nonetheless, to attenuate this choice, we also provide a `get` and `remove` operations with a *filter predicate*, allowing attribute-based filtering, and reducing the amount of transferred data (by filtering data where it is stored, before replying back).

In order for this mechanism to be decentralized and lightweight (in communication and state), we deliberately trade-off some structure(d) organization for a small amount of coordination. Thus, this mechanism is somewhat disorganized, and relies heavily on randomness properties.

5.4.1 Algorithm

Since each high-level (i.e., application) key can be partitioned, we need to have a way of identifying and locating these different partitions. Thus, each high-level key can have multiple low-level (i.e., internal) keys. To model this and identify these different keys, we resort to what we called *operation keys (opkeys)*. In PARSLEY, instead of using multiple hash functions, we salt the keys [132, 220]. We designate an indefinite amount of salts and number them from 0 to i. Then, each opkey contains the original (high-level) key provided by the application (e.g., a string), and its salt index or shard identifier (sid). By applying the hash function to the combination $\langle key, salt \rangle$, we get the corresponding key hash (used for DHT routing). Thus, an opkey is a tuple of the form[4]

$$\langle key, sid, hash \rangle$$

We also create the concept of a *shard index*. For each high-level key, a shard index holds the mapping between (known) sids and their corresponding key hashes. By storing the sids, we track which shards exist for a specific (high-level) key. Thus, a shard index is a tuple of the form

$$\langle key, [sid \mapsto hash] \rangle$$

In the next algorithms, we make use of the typical DHT procedure ROUTE(m, k), presented in Algorithm 2.2. It routes a message m to the owner of the specified key k.

Algorithm 5.2 gives an overview of our sharding mechanism. Periodically, during the group size check, the group leader runs the CHECKSTORAGEHOTSPOTS procedure (only if there was no split or merge). It starts by calling the GETHOTSPOTKEYS function (line 4), that returns a collection of all the locally stored opkeys considered to be hot-spots (see §5.4.2).

Afterwards, for each of those opkeys, the following is executed. First, we obtain the corresponding shard index for that key, and add a new shard (lines 6 and 7). This new shard will have the next available sid (for which the current group is not the shard's key owner). Next, the values mapping to this opkey are split in half, and the part destined for the new shard is removed from the local storage (lines 8 and 9). The DIVIDEVALUES function can take any approach, from simply random to solving something similar to the knapsack problem [168]. Currently, our implementation shuffles the values and divides them into two collections with roughly the same size (in bytes).

Then, three communication steps are performed. First, a new opkey is created for this new shard, and a NewShard message is sent to the key owner group with the partition values for it to store (lines 10–12). Next, the group leader that triggered this sharding sends a ShardUpdate message to all its group members, informing them about this new shard and the oids of the values to be removed locally (lines 13–15). Lastly, since we keep a proactive synchronization among the various shards of a key, a ShardUpdate message is also sent to all the other shards, but only with the newly added sid (lines 16–19).

When a peer receives a NewShard message (line 20), it first checks if the received opkey is already known (and has an associated shard index). If not (line 22), a new shard index is created with the received information. Otherwise, the existing shard index is updated with the received one (line 26). Then, the new values received in the message are stored locally (line 27).

[4]The last element, *hash*, is used just to save on computations, since it can be computed (lazily) from the other two elements.

Algorithm 5.2 PARSLEY's dynamic data sharding mechanism.

```
1: $store \leftarrow \emptyset$                                   ▷ key-value store [OpKey ↦ [Value]]
2: $shards \leftarrow \emptyset$                                  ▷ known shards [Key ↦ Shard Index]

3: procedure CHECKSTORAGEHOTSPOTS( )
4:     $hotspots \leftarrow$ GETHOTSPOTKEYS($store$)
5:     for all $opKey \in hotspots$ do
6:         $index \leftarrow shards[opKey.key]$
7:         $sid \leftarrow index$.ADDNEWSHARD( )
                                   ▷ divide values between the two shards and remove one of them
8:         $vals \leftarrow$ DIVIDEVALUES($store[opKey]$)
9:         $store[opKey] \leftarrow store[opKey] \setminus vals$
                                   ▷ send values to new shard group
10:        $newOpKey \leftarrow$ NEWOPERATIONKEY($opKey.key, sid$)
11:        $msg \leftarrow \langle$NewShard$, newOpKey, index, vals\rangle$
12:        ROUTE($msg, newOpKey.hash$)
                                   ▷ update my group peers about this new shard
13:        $oids \leftarrow [v.oid$ for $v \in vals]$
14:        for all $p \in group.view$ do
15:            SEND($\langle$ShardUpdate$, opKey, sid, oids\rangle, p$)
                                   ▷ update other shards about this new one
16:        for all $i \in index$.GETSHARDIDS( ) do
17:            if $i \neq sid \land i \neq opKey.sid$ then
18:                $msg \leftarrow \langle$ShardUpdate$, opKey, sid, \perp\rangle$
19:                ROUTE($msg, index.hash[i]$)

20: upon receive $\langle$NewShard$, opKey, sindex, vals\rangle$ from $src$ do
21:     $index \leftarrow shards[opKey.key]$
22:     if $index = \perp$ then
23:         $index \leftarrow$ NEWSHARDINDEX($sindex$)
24:         $shards[opKey.key] \leftarrow index$
25:     else
26:         $index$.UPDATE($sindex$)
27:     $store[opKey] \leftarrow store[opKey] \cup vals$

28: upon receive $\langle$ShardUpdate$, opKey, sid, oids\rangle$ from $src$ do
29:     $index \leftarrow shards[opKey.key]$
30:     $index$.ADDSHARD($sid$)
31:     $store[opKey] \leftarrow [v$ for $v \in store[opKey] : v.oid \notin oids]$
```

In turn, when receiving a ShardUpdate message (line 28), a peer simply adds the new sid to its shard index, and removes the values corresponding to the received oids.

On the opposite case, when an opkey is considered a *cold-spot*, the removal of a shard is analogous. When checking for hot-spot opkeys, the algorithm also checks for cold-spot ones. If so, it informs the other shards of such situation (and its group peers), removes its local shard, and transfers it to one of the other known shards chosen at random.

Note that the *primary shard* (i.e., shard 0) is treated as a special case. This is because, when a peer issues a `get` operation on some key, and it does not know if that key is sharded, it will always send the operation to the primary shard. So, even if the primary shard removes its content (for being a cold-spot), it will still keep the shard index to be able to redirect received operations.

As already mentioned, this sharding mechanism is somewhat disorganized and thus entails a small amount of coordination among shards. This presents a caveat. There may happen the case when two different shards partition the same key for the same next available sid at the same time (line 7). This does not affect the correctness of the mechanism. However, it entails more communication because both shards will send a

NewShard message to the owner of the new opkey, and in the next periodic check for hot-spots, that opkey will almost surely be partitioned again. This is a trade-off and the price to pay for the small coordination among shards.

If space is of concern, the shards map (line 2) can work as a cache of known sharded keys. Regarding the keys a group owns, the shards map will keep that information as long as the group owns them. However, information about other shards, that has been collected over time, can be kept using a known caching policy, e.g., LFU, LRU.

Naturally, we make this storage load balancing at the expense of data transfers (when sharding keys). However, if the popularity trend continues, and if the shards are already in place, the put operations will be automatically scattered among the shards. Thus, potentially reducing and amortizing the overall costs.

5.4.2 Defining Hot-Spots

One of the critical parts of this data sharding mechanism is to identify popular, or hot-spot keys. Thus, it is necessary to define the meaning of popular key. What is a popular key? To answer this question, Parsley allows the configuration of the definition of popular (or hot-spot) key, through what we call *hot-spot detectors*. The getHotSpotKeys function (line 4 in Algorithm 5.2) encapsulates that logic. It receives the current local store as a parameter, and returns a collection of opkeys considered to be hot-spots according to the implemented logic.

Currently, we offer four different hot-spot detectors out-of-the-box. Since we are storage-oriented, we take into account not only the amount of values per key, but also their size *in bytes*. We call *global size* to the sum of the size of all the values of the entire local store, i.e., $\textsc{bytes}(store)$. In turn, the *local size* refers to the sum of the values' size of a specific opkey k, i.e., $\textsc{bytes}(store[k])$.

Here, it is only possible (and it only makes sense) to partition an opkey if it has at least two values associated with it. Thus, all the different implemented alternatives have the following implicit restriction when checking an opkey k: $|store[k]| > 1$.

Absolute Amount (AA). An opkey k is considered an hot-spot if the number of values associated with it is greater than a configurable absolute value ρ, with $\rho \in \mathbb{N}_1$:

$$|store[k]| > \rho$$

Absolute Size (AS). An opkey k is considered an hot-spot if its local size is greater than a configurable absolute value ρ (in bytes), with $\rho \in \mathbb{N}_1$:

$$\textsc{bytes}(store[k]) > \rho$$

Relative Size (RS). An opkey k is considered an hot-spot if its local size is greater than a configurable percentage ρ of the global size, with $\rho \in]0,1[$, and the number of values associated with it is greater than a configurable absolute value ϕ, with $\phi \in \mathbb{N}_1$:

$$\textsc{bytes}(store[k]) > \textsc{bytes}(store) \times \rho \ \land \ |store[k]| > \phi$$

The second condition is necessary when there is only a small number of opkeys in the local store. In these cases, a small opkey can represent a large portion of the local store (or, in the extreme case, even its entirety). For instance, if the local store has only one opkey with two (small) values, this opkey will represent 100% of the store, thus making the first condition always true. Even though this opkey is small and not worth to be partitioned, it would be considered an hot-spot if not for the second condition.

Higher Than Average (HTA). An opkey k is considered an hot-spot if its local size is greater than a configurable percentage ρ *over* the average size per opkey of the local store or if this key is the only one in the local store, and the number of values associated with it is greater than a configurable absolute value ϕ, with $\phi \in \mathbb{N}_1$:

$$\left(\textsc{bytes}(store[k]) > \frac{\textsc{bytes}(store)}{|\textsc{keys}(store)|} \times (1+\rho) \ \lor \ |\textsc{keys}(store)| = 1\right) \land \ |store[k]| > \phi$$

Note that, in this case, ρ can also take the value zero or even negative values. Thus, it is possible to refer to values equal to or less than the average.

If this key is the only one in the local store, it will be the average and the first condition would not be triggered (when $\rho > 0$), justifying the need for the disjunction. The outer condition is here for the same reason as the previous hot-spot detector—to ensure that this key's partitioning is worthwhile.

5.4.3 DHT Operations

Because of this sharding mechanism, DHT operations have to be slightly modified. Algorithm 5.3 depicts an overview of the modifications.

When issuing a put operation, it is sent to one of the locally known shards, chosen at random (procedure PUT in lines 1–8). In case no shard is known, it is sent to the primary shard. If more than one shard is known, preferably, it should be sent to any shard but the primary (to alleviate the load on that shard and promote overall load balancing). As a reply, the issuer receives the shard index known locally by the peer that processed the operation, that is stored for future use (lines 9–15).

Both get and remove operations work the same way, and it is the same for both their versions (i.e., with or without the filter predicate), requiring it to be sent to all the existing shards (procedure GET in lines 16–24). On the receiving end, the operation receiver replies back with the requested values and the locally known sids. Then, the issuer gathers those received sids, and triggers the sending of the same ongoing operation to those previously unknown shards, if any (lines 30–33). In the case of get operations, the upper layer is notified of the operation result when there are no more ongoing requests and replies have been received from all known shards (line 34–35).

Algorithm 5.3 Parsley's modified DHT operations.

```
 1: procedure PUT($k, v$)
 2:     $index \leftarrow shards[k]$
 3:     if $index = \bot$ then
 4:         $index \leftarrow$ NEWSHARDINDEX($k$)
 5:         $shards[k] \leftarrow index$
 6:     $sid \leftarrow index$.GETRANDOMSHARDID( )          ▷ preferably, not the primary shard
 7:     $opKey \leftarrow$ NEWOPERATIONKEY($k, sid$)
 8:     ROUTE(⟨Put, $opKey, v$⟩, $opKey.hash$)

 9: upon receive ⟨PutReply, $sindex$⟩ from $src$ do
10:     $index \leftarrow shards[sindex.key]$
11:     if $index = \bot$ then
12:         $index \leftarrow$ NEWSHARDINDEX($sindex$)
13:         $shards[sindex.key] \leftarrow index$
14:     else
15:         $index$.UPDATE($sindex$)

16: procedure GET($k$)
17:     $index \leftarrow shards[k]$
18:     if $index = \bot$ then
19:         $index \leftarrow$ NEWSHARDINDEX($k$)
20:         $shards[k] \leftarrow index$
21:     keep track of ongoing requests
22:     for all $sid \in index$.GETSHARDIDS( ) do
23:         $opKey \leftarrow$ NEWOPERATIONKEY($k, sid$)
24:         ROUTE(⟨Get, $opKey$⟩, $opKey.hash$)

25: upon receive ⟨GetReply, $opKey, vals, sids$⟩ from $src$ do
26:     keep track of ongoing/finished requests
27:     save received (partial) results (i.e., $vals$)
28:     $index \leftarrow shards[opKey.key]$
29:     $index$.UPDATE($sids$)
30:     $newSids \leftarrow$ FILTERNEWSIDS($sids$)          ▷ get sids that were unknown before this reply
31:     for all $sid \in newSids$ do
32:         $newOpKey \leftarrow$ NEWOPERATIONKEY($opKey.key, sid$)
33:         ROUTE(⟨Get, $newOpKey$⟩, $newOpKey.hash$)
34:     if no more ongoing requests then          ▷ get operation finished
35:         NOTIFYUPPERLAYER($opKey.key, results$)
```

5.5 Evaluation

In this section, we present the experimental results that showcase and validate our proposed mechanisms. We implemented a prototype of Parsley in PeerSim [180] and use it to experimentally validate its design, detailing and characterizing its behavior in different large scale scenarios.

Comparisons of group-based overlays against other DHTs and different replication mechanisms have already been done in previous work in the literature [139, 150, 192, 193]. Thus, here the focus of this evaluation section is on the differentiating factors of our solution, namely its PPR and data sharding mechanisms.

In this experimental work, we assess: 1) Parsley's resilience to churn, and the benefits of PPR and its corresponding overheads (§5.5.2); 2) the benefits and overheads of storage load balancing through our dynamic data sharding mechanism (§5.5.3); and 3) the overlay's management overhead (§5.5.4). Additionally, we present an overlay characterization regarding the group size parameters, in Appendix A (justifying some of the values used to configure our system).

5.5.1 Experimental Setup

We conducted an extensive experimental evaluation of Parsley in the PeerSim simulator using its event-driven engine. All results presented in this section are an average of results extracted from 20 independent executions for each data point.

PeerSim has a virtual clock that coordinates the delivery of events to peers. In the simulator, one cycle represents 1 000 time units (TUs), thus each TU can be seen as one millisecond (and a cycle as one second). To experiment in large-scale scenarios, unless stated otherwise, the experiments were conducted in a system comprised by 10 000 peers, and was populated with 50 000 values distributed among 10 000 keys (resulting in an average of five values per key). Values are assigned to keys following specific distributions, ensuring that every key has at least one value assigned. Keys are chosen uniformly at random from the key space, and values' size follows a normal distribution with a mean value of 5 MB and a standard deviation of 1 MB (representing reasonable sized files, such as photos taken by a smartphone) [95, 218]. In the end, yielding a total of around 250 GB.

To configure Parsley, we first did an overlay characterization study regarding the group size parameters, reported in Appendix A. There, we analyze several metrics and lay our rationale for the chosen group size parameters. Thus, throughout this section, group size thresholds are set to $l = 4$, $l' = 5$, $h' = 10$, and $h = 11$. The maximum load threshold was set to 1.75, in order to represent a good balance between group size and amount of stored data. The group maintenance frequency was set to one second, with a probability of 10% (i.e., $\gamma = 0.1$) [150], thus resulting in one maintenance procedure being triggered each 10 seconds on average (by each peer). The periodic group size check was executed with a frequency uniformly distributed between two and four seconds.

5.5.2 Churn & Peer Relocation

Any peer-to-peer (P2P) overlay targeting large-scale and dynamic environments should be able to handle churn efficiently. To stress Parsley's capability of handling churn, and assess the benefits of our PPR mechanism, in this section, experiments were conducted as follows. The overlay was initialized by having peers join the system one at a time. After a stabilization period, churn was induced during a period of 60 simulation cycles. Every other cycle during the churn period, c peers are removed simultaneously. When the churn period is over, another stabilization period is executed, and the simulation halts. The first peer in the overlay generates and stores all the key-value pairs at the beginning of the simulation. In these experiments, c takes the values described in Table 5.2. We also mention the percentage of peers that are removed from the overlay at each churn *moment*, and at the *end* of the churn period (both regarding the overlay's initial number of peers). For instance, with $c = 200$, at the *end* of the churn period, this amounts to 60% of the initial number of peers. We refer to the end percentage in the plots. By using increasing amounts of churn, we can assess how the overlay behaves with increasing dynamics in system filiation. Values have been assigned to keys following a uniform distribution.

Table 5.2: Configuration of parameter c (i.e., churn) in Parsley.

c (peers)	17	33	67	100	133	167	200	233	267	300	317
% (moment)	0.17	0.33	0.67	1	1.33	1.67	2	2.33	2.67	3	3.17
% (end)	5	10	20	30	40	50	60	70	80	90	95

Here, we use two different scenarios. First, one where peers leave the system and no new peers enter—which we refer to as *exit-only*. This scenario allows us to observe the behavior of the overlay in extreme scenarios, where peers leave the system and there are no new peers that can replace the ones that left, resulting in an abrupt decrease in network size. Secondly, we use another scenario where peers leave the system and the same amount of new peers join the overlay—which we refer to as *enter-exit*. This scenario allows us to verify the behavior of the overlay in churn-intensive situations, but having new peers entering the system to offset the ones leaving, maintaining a stable network size throughout the simulation.

The periodic relocation of peers is checked every 20 seconds, and the relocation cool down period is also 20 seconds. So as not to interfere with the results, we also disabled the coercive merge feature (§5.3.3.5).

Additionally, groups are divided into two sets (hot and cold), defined by a distribution ratio. In these experiments, we set the distribution ratio to be 50%, thus both sets have the same number of groups. Then, peers in the hot set have a configurable probability ϵ of being chosen to leave the system (i.e., churn), while peers in the cold set have the complementary probability (i.e., $1-\epsilon$). Unless stated otherwise, in these experiments, we set probability $\epsilon = 0.8$. Every step, before removing peers from the system, the sets are updated (e.g., update groups' membership, remove merged groups or that no longer exist), and are also rebalanced to maintain the distribution ratio (e.g., add new groups created through splits). These sets are maintained throughout the simulation.

In this section, we compare Parsley using the following configurations:

No PPR (NPPR in the plots) - with the peer relocation mechanism disabled (which can be seen as similar to Rollerchain [192]);

Push - with the peer relocation mechanism using only push requests, i.e., only larger groups try to give some of their peers to smaller groups;

Pull - with the peer relocation mechanism using only pull requests, i.e., only smaller groups try to request peers from larger groups; and

Full PPR (FPPR in the plots) - with the peer relocation mechanism fully enabled (i.e., using both push and pull requests).

All the plots hereafter depict data collected from the start of the churn period until the end of the simulation.

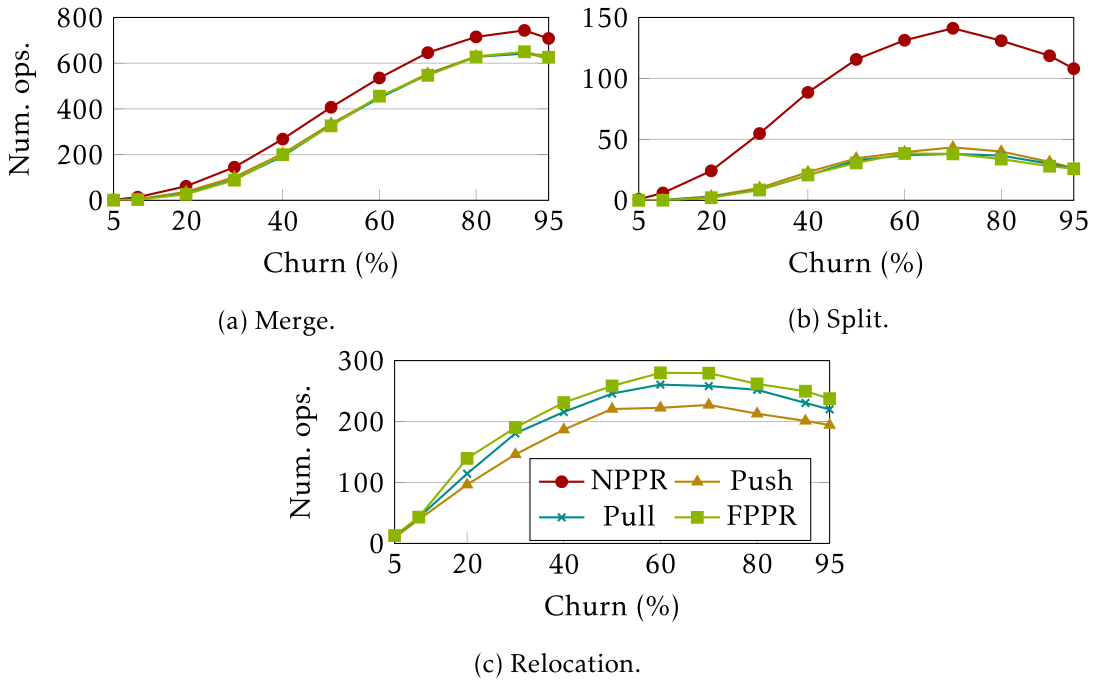
Remember that MobiStore (described in §5.2) is a one-hop DHT. This means that, over time, each group will tend to know every other group in the overlay. Thus, in this type of DHTs, peers will have much more information at their disposal, naturally influencing the decisions they make. Here, comparing MobiStore and PARSLEY will never be an apples to apples comparison, being merely a hint. The most important argument we want to make is to verify that the PPR mechanism makes sense and is beneficial in a typical DHT (i.e., without being one-hop), like PARSLEY.

5.5.2.1 Exit-Only

In this scenario, there are no peers entering the overlay. That is, in every churn period, c peers leave the system and are not replaced. This results in an abrupt decrease in the overlay size. Since peers in the hot set have a higher probability of leaving, those groups will loose peers much quicker. Thus, this can also cause entire groups to be removed at once from the overlay. Note that, with $c = 317$ peers, at the end of the churn period, 95% of the peers in the overlay were removed from the network (in a period of 60 seconds)—an extreme churn scenario.

Figure 5.6 depicts the amount of merge, split, and relocation operations executed during the simulation. Naturally, the number of executed operations follows the increase in the amount of churn. As more peers leave the system, groups have to accommodate those changes and merge (and sometimes also split afterwards). Nonetheless, there are some occasions when peers are relocated from larger into smaller groups.

Regarding the number of merge operations, in Figure 5.6a, at a glance, it may not



(a) Merge.

(b) Split.

(c) Relocation.

Figure 5.6: Exit-only topology operations in PARSLEY.

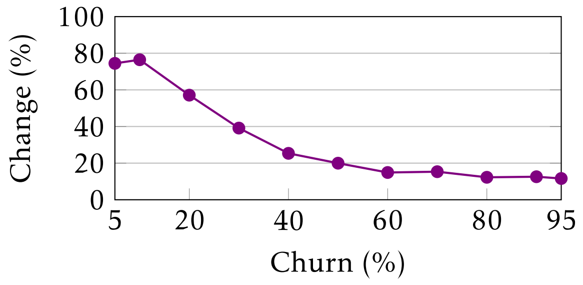


Figure 5.7: Relative change in the number of merge operations between No PPR and Full PPR, for the exit-only scenario in Parsley.

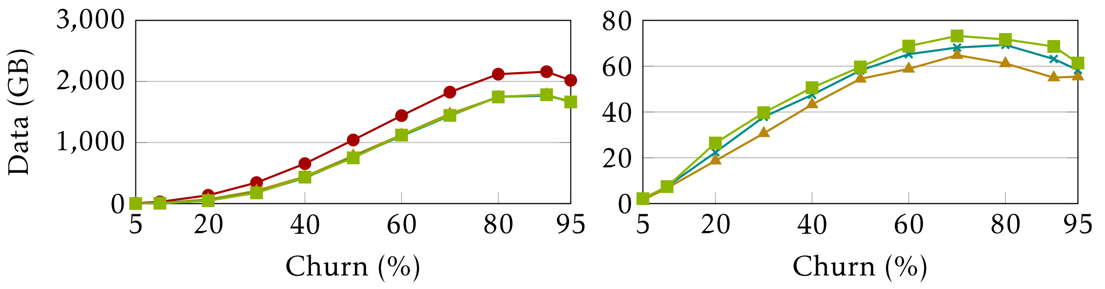
seem like it, but the difference between the plots for NPPR and FPPR decreases as the amount of churn increases. Figure 5.7 denotes the relative change [271] between NPPR and FPPR as a percentage, i.e., $(\frac{v^{NPPR}-v^{FPPR}}{v^{FPPR}}) \times 100$. For instance, in this case, with $c = 100$ or 30% churn, NPPR executes 40% more merge operations than FPPR. However, NPPR always requires more merge operations, since each time a group reaches its minimum size threshold, there is no other option besides merging. Here, all the alternatives with some kind of peer relocation take advantage of this feature to reduce the number of required merge operations. Nonetheless, as there are only peers leaving the overlay, groups will keep shrinking until reaching a point where they do not know any other too big groups from where to request peers. In the end, they also have no other option but to merge.

In a similar note, in Figure 5.6c, we see that the number of peer relocation operations reaches a plateau with large amounts of churn. This is because in this scenario there are no peers entering the overlay. They are only leaving. Thus, as the amount of churn grows, the system starts to decrease in size, and there are less and less opportunities for peer relocations (i.e., there are less too big groups).

Regarding Figure 5.6b, we can see that the number of split operations in NPPR grows very quickly with the amount of churn. In turn, all the other plots with some kind of peer relocation grow slowly as the amount of churn increases. This proves the effectiveness of our simple optimization to avoid merge-caused splits (§5.3.3.6). In the alternatives with peer relocation, when a group receives a merge request, if it detects that the merge would result in a split afterwards, it cancels the merge and relocates some of its peers into the previously merging group. Thus, working to reduce the number of required split operations by a great amount.

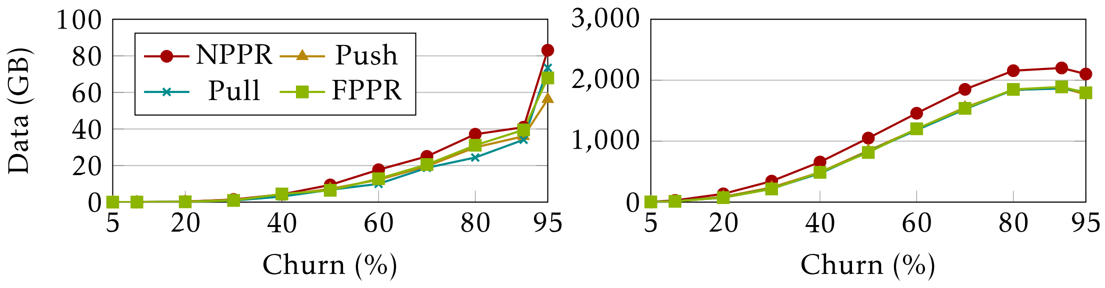
In turn, Figure 5.8 depicts the amount of data items transferred (in GB) as a result of merges, relocations, group maintenance, and the accumulated total. Directly, we can see that the total is dominated by the amount of data transferred due to merge operations (Figure 5.8a), revealing the importance of trying to reduce the number of required merges. Also, Figure 5.8c shows that the amount of data items transferred due to group maintenance is not very relevant, and is very similar across all alternatives.

Figure 5.8d shows us that the alternative with NPPR requires the transfer of more



(a) Merge.

(b) Relocation.



(c) Maintenance.

(d) Total.

Figure 5.8: Exit-only data transfers in Parsley.

data across all the churn values. Since all the other alternatives employ some kind of peer relocation, they reduce the number of merge operations (as shown in Figure 5.6), and consequently the amount of data transfers. Regarding FPPR, for small amounts of churn, it achieves a reduction of as much as 55% in data transfer, when comparing with NPPR (as illustrated in Figure 5.9). For large amounts of churn, since there are less opportunities for peer relocations, the difference reaches as much as 15%. In the end, since merge operations dominate the total data transfers, the savings in data transfers of FPPR also decreases and the amount of churn grows.

Figure 5.8b follows the rationale in Figure 5.6c, where Push has less opportunities for peer relocations, but achieves around the same savings in data transfers. This is something to take into account when configuring our system. The Push alternative is able to achieve around the same data transfers savings with less peer relocations.

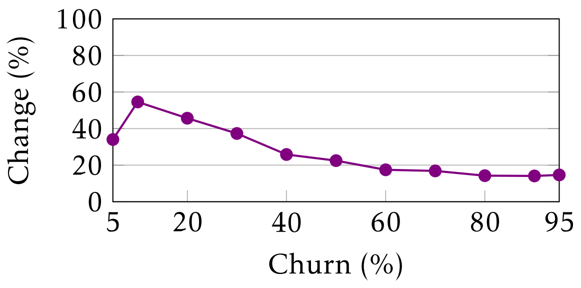
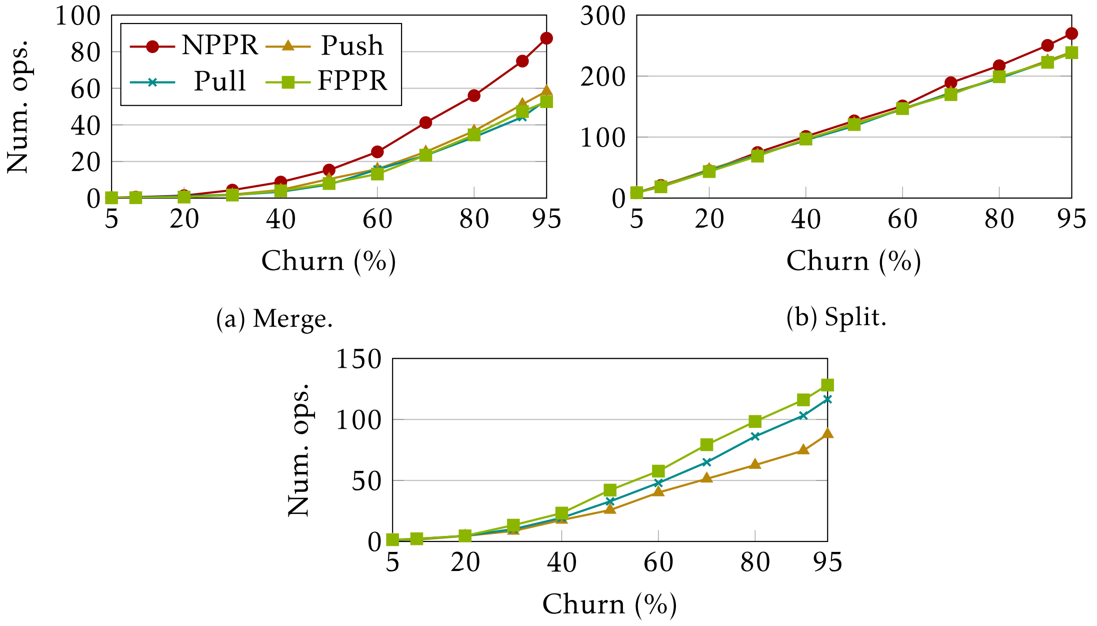


Figure 5.9: Relative change in the amount of total data transfers between No PPR and Full PPR, for the exit-only scenario in Parsley.



(a) Merge.

(b) Split.

(c) Relocation.

Figure 5.10: Enter-exit topology operations in Parsley.

5.5.2.2 Enter-Exit

In this scenario, there are peers entering the overlay as others leave, by the same amount. Thus, these represent churn-intensive situations, but with a stable network size.

Figure 5.10 depicts the amount of merge, split, and relocation operations executed during the simulation. Similarly to the *exit-only* scenario, the number of executed operations follows the increase in the amount of churn. However, here, the absolute values are much smaller for both merge and relocation operations (Figures 5.10a and 5.10c). For splits, it grows in a (sub)linear proportion to the amount of churn, with NPPR executing more splits (Figure 5.10b). Since peers enter the overlay as other fail, they end up filling the voids. Thus, these operations are needed to accommodate the rapid changes in the network, but by a small amount when compared to the *exit-only* scenario.

Regarding merge operations, in Figure 5.10a, we can see that NPPR requires always more operations across all churn values. Despite having the same amount of peers entering and leaving the overlay, these somewhat rapid changes to the groups' memberships triggers those merge operations. Nevertheless, after the system quiesces, those memberships stabilize, requiring almost eight times less merge operations as in the *exit-only* scenario. Also, the difference between NPPR and FPPR does not vary as much as in the previous *exit-only* scenario. Figure 5.11 shows the relative change as a percentage. Here, the average difference reports around 47% less merge operations for FPPR, when compared with the NPPR configuration.

The number of peer relocations, in Figure 5.10c, grows slowly with the amount of churn. Since peers enter the overlay, groups are able to keep their sizes relatively the

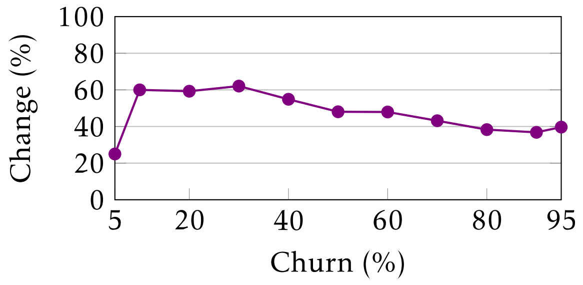
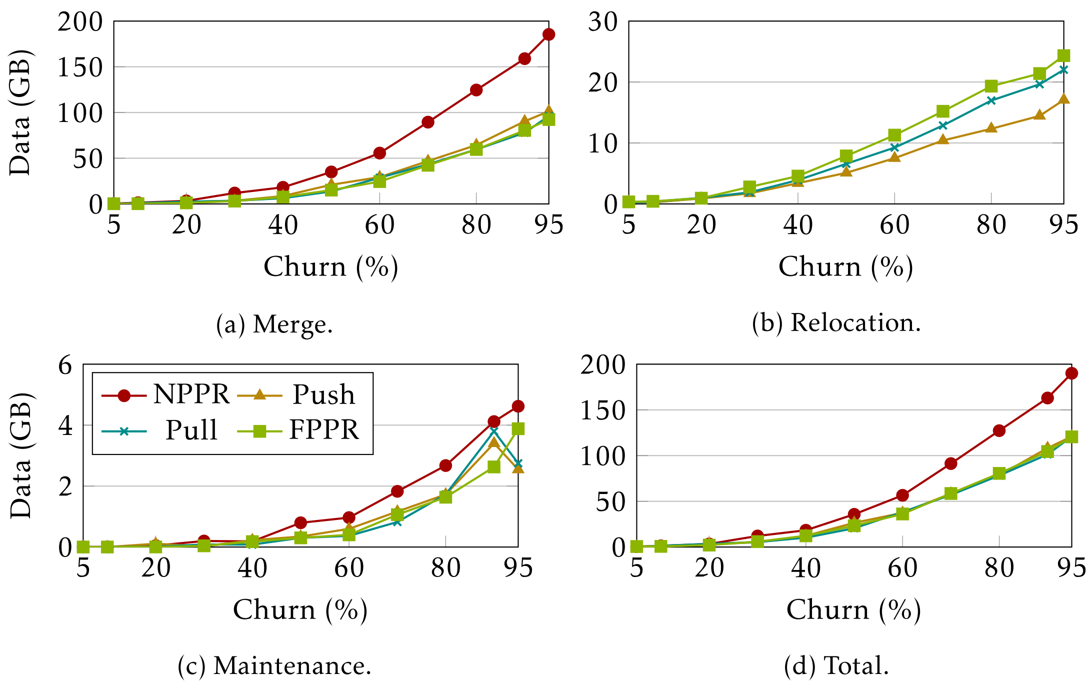


Figure 5.11: Relative change in the number of merge operations between No PPR and Full PPR, for the enter-exit scenario in PARSLEY.

same (see §5.5.2.3). Thus, not requiring a great amount of relocation operations. Nevertheless, since peer relocations are validated by a peer of the group where the relocation is to happen (before it actually happens), this decreases the amount of unnecessary relocations in this scenario. Once again, the Push alternative is able to achieve almost the same savings in the number of merge operations, requiring less peer relocations, when compared with FPPR.

Figure 5.12 depicts the amount of data transferred as a result of merges, relocations, group maintenance, and the accumulated total. First, similarly to the *exit-only* churn scenario, the total is dominated by the data transfers caused by merge operations (Figure 5.12a). Additionally, Figure 5.12c illustrates that the data transferred due to group maintenance is not very relevant (accounting for around 2% of the total transfers), being similar across all alternatives.



(a) Merge.

(b) Relocation.

(c) Maintenance.

(d) Total.

Figure 5.12: Enter-exit data transfers in PARSLEY.

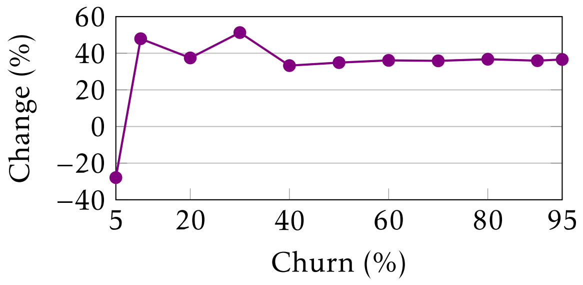


Figure 5.13: Relative change in the amount of total data transfers between No PPR and Full PPR, for the enter-exit scenario in Parsley.

Also in this scenario, Figure 5.12d shows us that NPPR requires much more data transfers than all the other alternatives, across the majority of the churn values. In turn, the peer relocation mechanism shows its operation, effectively reducing the overall amount of data transfers in the system.

For low churn, and taking into account that peers are also entering the overlay, not many peer relocations are actually required. Thus, FPPR may trigger some peer relocations that, in the end, would not be necessary. This is shown in Figure 5.13. With 5% churn, FPPR transfers 28% more data than NPPR. For such small values of churn, the amount of data transferred is very small, and as such, the relative difference between the two is more pronounced. Here, the case is that some peer relocations may be unnecessary, and they are triggered because of momentary inconsistencies in groups' memberships. On the other hand, for all the other churn values, FPPR achieves a more stable savings in data transfers, with an average around 39% (i.e., less data transferred than NPPR).

Lastly, Figure 5.12b reiterates the fact that the Push alternative is able to achieve a similar data transfer savings to the FPPR configuration, while at the same time requiring less peer relocation operations.

5.5.2.3 Group Size & Data Loss

The values for both the amount of lost keys and group size across all alternatives present a negligible difference. Thus, we report them as an average of all the alternatives.

Regarding group size, Figure 5.14 shows the average group size at the end of the simulation. For the *enter-exit* scenario, there is almost no difference in group size. From the first churn value to the last, there is a decrease of around one. This can be explained by the entering of new peers in the overlay, filling the gaps left by the failing peers. However, in the *exit-only* scenario, we can see a more significant decrease in group size as the amount of churn increases. Once again, since there are no new peers entering the overlay, groups can only merge to try to sustain those failures (or sometimes relocate peers). Nonetheless, as the amount of churn increases, groups start to get smaller. From 5% to 95% churn, we see a decrease of more than three.

Figure 5.15 depicts the percentage of lost keys at the end of the simulation, for the

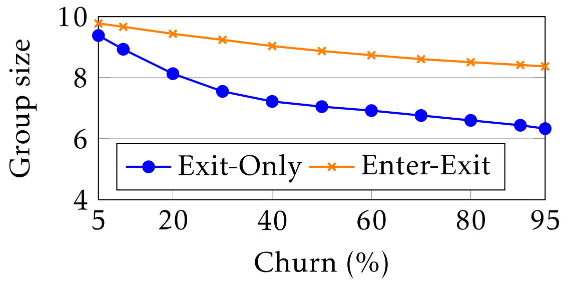


Figure 5.14: Average group size for both churn scenarios in Parsley.

various churn values. Here, we can see that in the *enter-exit* scenario, there is practically no data loss across all churn values (reaching a loss of less than 0.4% with 95% churn). This is explained by the entry of new peers in the system, offsetting the exit of the failing peers. Moreover, the *new peer acceptance logic* (§5.3.3.2) favors peers to be accepted in smaller or overloaded groups, increasing the overlay reliability. In turn, in the *exit-only* scenario, data loss accompanies the increase in churn. Since peers only leave, and no new peers enter the overlay, groups start to become smaller as time passes. In fact, because of the hot set logic, some peers have a higher probability of leaving, and some groups will loose peers very quickly. Thus, this can also possibly cause entire groups to be removed at once from the overlay. If this happens, since there was no time to do the proper backups, the data stored by those groups will be lost. With 95% churn, there is a loss of around 73% of all the keys. However, note that, this represents losing 95% of the overlay total peers. Even losing such a large number of peers, the system is able to retain 27% of the stored keys. In this scenario, data loss starts to become an issue at around 60% churn (i.e., a loss of 60% of the overlay peers), with a data loss of almost 3% of all the keys.

5.5.2.4 Balanced Hot & Cold Sets

In the previous experiments, groups were divided into two sets—hot and cold. Both sets had the same size, but peers from groups in the hot set had 80% probability of being chosen to leave the system (the parameter ϵ described at the begging of this section). At the same time, peers from groups in the cold set had only 20% probability of being

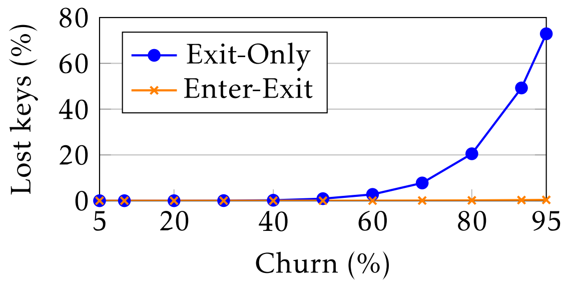
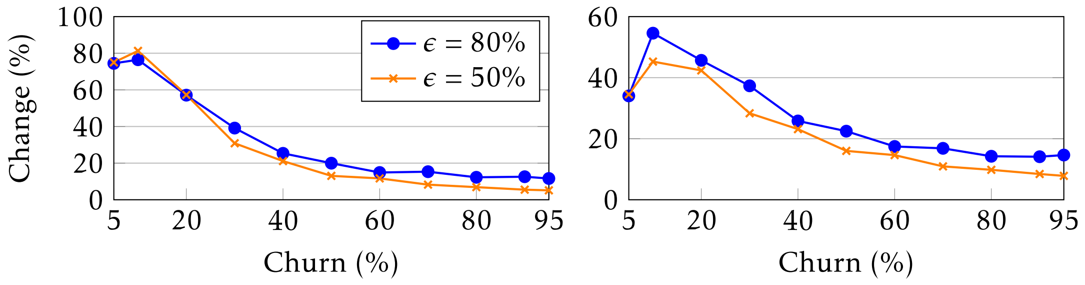


Figure 5.15: Average percentage of lost keys for both churn scenarios in Parsley.



(a) Merge operations.

(b) Total data transfers.

Figure 5.16: Relative change between No PPR and Full PPR, for the exit-only scenario in Parsley.

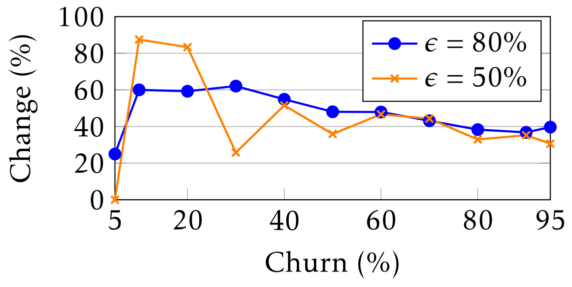
chosen. This is an asymmetric scenario that helps to showcase the churn resilience of Parsley and the advantages of PPR. Nonetheless, this can also be verified in a balanced scenario where all peers have the same probability of leaving the system, although in a less conspicuous way. Next, we show some of the most important differences between these two scenarios: $\epsilon = 80\%$, and $\epsilon = 50\%$.

Figure 5.16 depicts the relative change in the amount of merge operations and total data transfers between NPPR and FPPR, for the *exit-only* scenario. Here, we can see that the difference between these two alternatives follows the same trend, but is less pronounced when $\epsilon = 50\%$. This is due to the balanced hot and cold sets. With $\epsilon = 50\%$, all peers have the same probability of leaving the overlay, and thus this probability is scattered among many more peers than before.

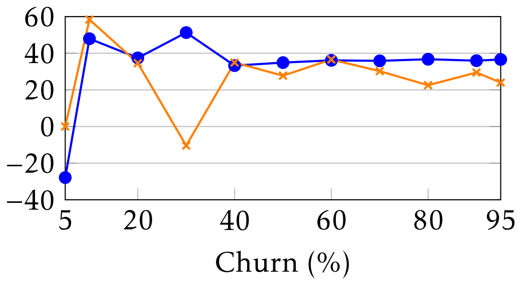
Regarding the *enter-exit* scenario, Figure 5.17 shows the relative change in the amount of merge operations and total data transfers between NPPR and FPPR. As in the previous figure, for large churn values, these plots show us that the difference between the two alternatives is less pronounced when $\epsilon = 50\%$. However, for smaller churn values (until around 40% churn), the difference is quite erratic when $\epsilon = 50\%$. This is due to the fact that, in this scenario, much less peer relocations are actually required. Thus, some of the executed relocations were in fact not necessary. Adding to that, the absolute values in these cases are very small, hence the relative difference between the two is more sensitive to small variations, and thus is more pronounced in the plots.

Figure 5.18 shows the total data transfers for the *enter-exit* scenario, when $\epsilon = 50\%$. When comparing with Figure 5.12d, we can see that in this scenario much less data transfers are required—around less than half of when $\epsilon = 80\%$. And, once again, here, the difference between the NPPR and FPPR alternatives is less pronounced. This is due to the fact that, in this scenario, all peers have the same probability of leaving the overlay. Thus, requiring less effort from all the overlay resilience mechanisms, and presenting more opportunities for peer relocations.

Figure 5.19 illustrates the percentage of lost keys at the end of the simulation, for the *exit-only* scenario. Here, the data loss metric follows the increase in the amount of churn,



(a) Merge operations.



(b) Total data transfers.

Figure 5.17: Relative change between No PPR and Full PPR, for the enter-exit scenario in PARSLEY.

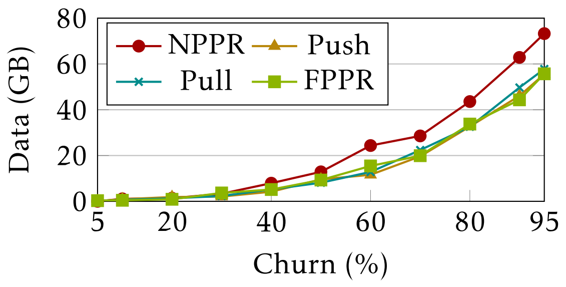


Figure 5.18: Total data transfers, for the enter-exit scenario with $\epsilon = 50\%$ in PARSLEY.

because there are no new peers entering the overlay, and thus entire groups can disappear at once. However, with $\epsilon = 50\%$, since all peers have the same probability of leaving the overlay, the probability of losing keys ends up being diluted evenly among all the peers/groups, thus being less pronounced. With 95% churn and $\epsilon = 50\%$, there is a loss of around 43% of all the keys, representing a decrease of 30% when compared to $\epsilon = 80\%$. Remember that 95% churn represents a loss of 95% of the overlay total peers, and even so the overlay manages to retain 57% of the stored keys. In this situation, data loss starts to become an issue at around 70% churn (i.e., a loss of 70% of the overlay peers), with a loss of almost 2% of the keys.

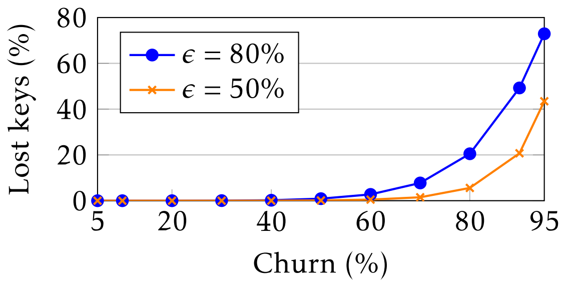


Figure 5.19: Average percentage of lost keys, for the exit-only scenario in PARSLEY.

Table 5.3: Impact of the zipfian distribution parameter in the probability of the most popular item (from a total of 10 000 items drawn 50 000 times).

Skew	0	0.5	0.75	0.99	1.25
Probability	$\approx 0.03\%$	$\approx 0.5\%$	$\approx 2.5\%$	$\approx 8\%$	$\approx 19\%$
Num. values	≈ 15	≈ 250	$\approx 1\,250$	$\approx 4\,000$	$\approx 9\,500$

5.5.3 Load Balancing Storage Hot-Spots

In some scenarios Parsley targets, like indexing, a few DHT keys may become very popular, thus overloading the owners of those keys. As such, in these situations, overlays should be able to adapt to that skewed popularity. To stress Parsley's capability of handling these cases, and assess the benefits of our dynamic sharding mechanism, in this section, experiments were conducted similarly to the previous one. The overlay was initialized by having peers join the system one at a time. After a stabilization period, peers start to issue put operations, storing the key-value pairs in the DHT. Put operations are issued by every peer, starting randomly in the first second after the stabilization period, and then with a frequency of 20 seconds, until there are no more key-value pairs to insert (i.e., each peer issues an average of 5 put operations). After the key-value pairs are all inserted, another stabilization period is executed, and the simulation halts.

We have assigned values to keys following two different distributions[5]: 1) uniform; and 2) zipfian. The uniform distribution provides a baseline for comparison. In turn, the zipfian distribution portrays the overlay operation in scenarios where some keys are much more popular than the rest, thus becoming overloaded. Hereafter, when mentioning a *skew of 0*, we are referring to the uniform distribution.

For the zipfian distribution, its parameter has implications in the skewness of the items' popularity. Table 5.3 shows the approximate observed impact of the distribution parameter specifically on the popularity of the most popular item (i.e., the DHT key with the most values assigned to it). In an exactly uniform assignment of 50 000 values among 10 000 keys, all keys would have the same probability of 0.01%, i.e., each key would have five values assigned.

To detach the load balancing properties of Parsley from other aspects of the overlay operation, we evaluate the behavior of our dynamic data sharding mechanism using stable topologies (i.e., without churn).

In this section, we present results for all the provided hot-spot detectors (§5.4.2), namely Absolute Amount (AA), Absolute Size (AS), Relative Size (RS), and Higher Than Average (HTA). Table 5.4 shows the values used to configure the parameters of each of the hot-spot detectors. We configure their parameters according to the defined skew, guided by the impact in the most popular key. For instance, with a skew of 1.25, the most popular key will have roughly 9 500 values (Table 5.3), thus for AA, using $\rho = 5$ would

[5]These distributions' implementation was taken from the Yahoo! cloud serving benchmark (YCSB) [61] repository: `https://github.com/brianfrankcooper/YCSB`.

Table 5.4: Configuration parameters for the hot-spot detectors in Parsley.

(a) AA.

Skew	ρ
0	3, 5, 8, 10, 13
0.5	5, 10, 20, 40, 80
0.75	
0.99	20, 40, 80, 160, 320
1.25	40, 80, 160, 320, 640

(b) AS (in megabytes).

Skew	ρ
0	15, 25, 40, 50, 65
0.5	25, 50, 100, 200, 400
0.75	
0.99	100, 200, 400, 800, 1 600
1.25	200, 400, 800, 1 600, 3 200

(c) RS.

Skew	ρ	ϕ
0		
0.5		5
0.75	0.1, 0.2, 0.4, 0.6, 0.8	
0.99		20
1.25		40

(d) HTA.

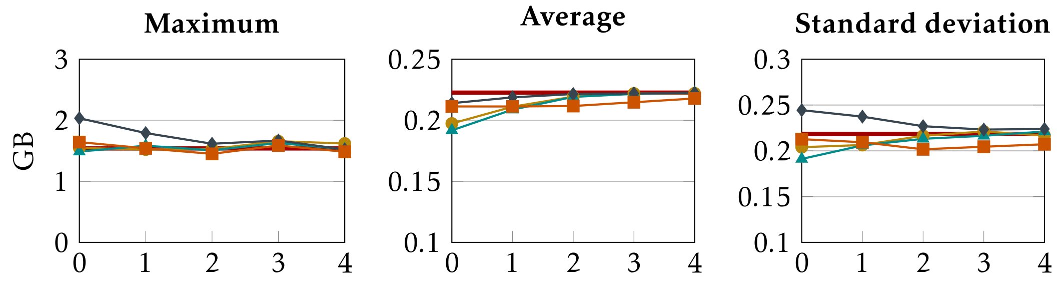
Skew	ρ	ϕ
0		
0.5		5
0.75	0.1, 0.3, 0.5, 0.7, 0.9	
0.99		20
1.25		40

entail almost 2 000 shards for this key. To avoid such an excessive partitioning of keys, we adapt the configuration parameters to the skew value. Hence, for increasing values of skew, we use correspondingly increasing values for the parameters (namely for the ρ parameter of AA and AS, and the ϕ parameter of RS and HTA).

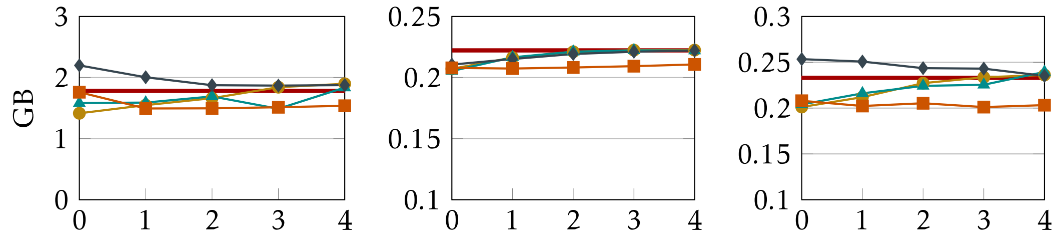
Additionally, we experiment with five different parameter values for each skew value, from less restrictive (*parameter 0* in the plots) to more restrictive values (*parameter 4* in the plots). Here, less restrictive parameter values will entail more popular keys, while in turn more restrictive values entail less popular keys (i.e., only the keys with many assigned values will be considered hot-spots).

Figure 5.20 shows some metrics regarding the keys (and corresponding values) stored by each group at the end of the simulation—that we refer to as *per group state*—, for each skew value. The left column shows the maximum per group state, while the middle column shows the average per group state, and the right column shows the corresponding standard deviation. In the plots hereafter, OFF refers to Parsley with the sharding mechanism disabled. For all skew values, the minimum per group state observed in the experiments was always zero, due to the fact that there were always some groups that did not store any keys. We observed an average of less than 8% of the groups in this situation, across all experiments.

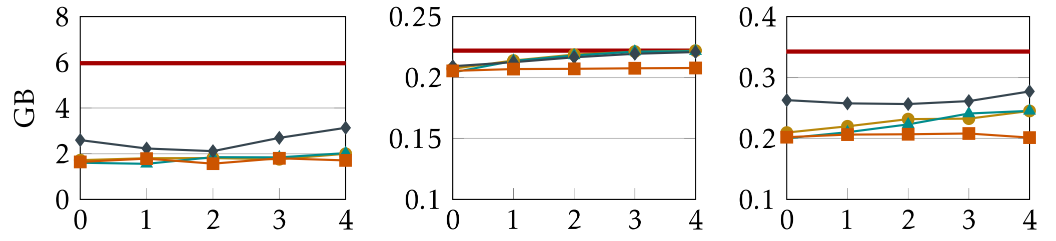
From the middle column in Figure 5.20, we can see that the average state per group is roughly the same for all configurations and across all the skew values—around 220 MB. This is expected, as the total amount of data stored in the overlay does not change, and neither the amount of peers/groups. As such, in this scenario, the average per group state should remain relatively unchanged across all experiments. In turn, Parsley's sharding mechanism largely influences both the maximum and standard deviation of



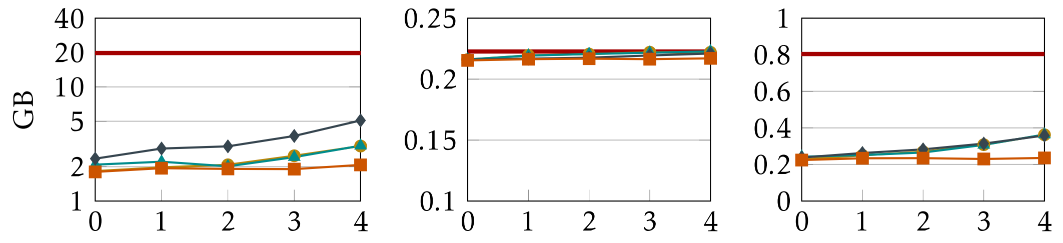
(a) Skew 0.



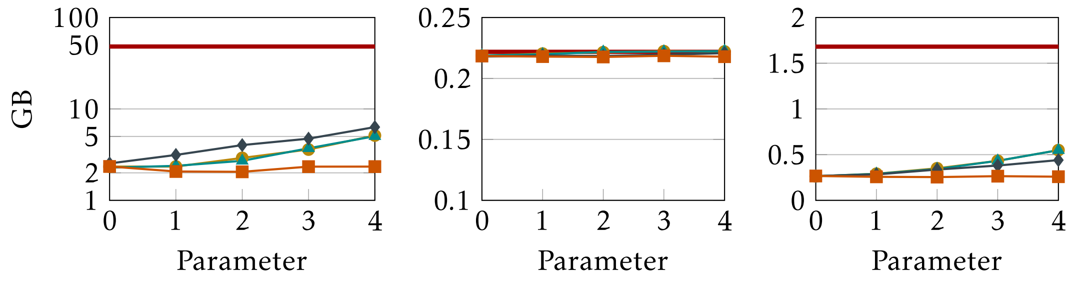
(b) Skew 0.5.



(c) Skew 0.75.



(d) Skew 0.99.



(e) Skew 1.25.

Figure 5.20: Per group state in PARSLEY (left to right: maximum, average, standard deviation).

the per group state, as shown in the left and right columns of Figure 5.20, respectively. Parsley is able to reach a reduction of up to 95% for the maximum per group state, and up to 85% for the standard deviation, when comparing with the system without sharding mechanism. This is achieved by partitioning keys with many assigned values into smaller portions (i.e., shards), and scattering them among several other groups in the overlay, effectively balancing the storage load.

With a small skew in the stored data, all configurations present roughly the same results in each of the three metrics. We can verify that small variation in Figures 5.20a and 5.20b, showing plots for skew values of 0 and 0.5, respectively. This is somewhat expected, since an uniform distribution (or with a very small skew) already does a kind of natural load balancing by itself. Consequently, all the sharding done to keys does not achieve any extra (meaningful) load balancing.

With a skew of 0, AA and AS are able to achieve the lowest average (and also the lowest standard deviation) from all the scenarios, for the less restrictive parameter (i.e., parameter 0)—at around 195 MB. This is due to the fact that with this parameter, keys are partitioned at a very fine-grain level. In this case, these hot-spot detectors identify keys as being hot-spots when one has more than three values or more than 15 MB assigned, respectively (Tables 5.4a and 5.4b). However, note that we are using 50 000 values assigned to 10 000 keys, resulting in an average of five values per key (the value used as parameter 1 for skew 0 in AA). Thus, with this configuration, we are partitioning keys below the average number of values per key, i.e., we are over-partitioning keys. By partitioning keys in such small shards, we are able to scatter them a little bit more among the overlay groups, and consequently reach a lower average per group state. As the parameter becomes more restrictive, less keys are considered hot-spots, and thus less keys are partitioned. As a consequence, the results get closer to the system without sharding—OFF in the plots.

Still in the same figures, regarding the maximum and standard deviation of the per group state, we can see that all alternatives behave more or less the same for all parameters. The exception is RS. For the less restrictive parameters, it achieves larger values than any other alternative, although with a small difference—less than 500 MB for the maximum per group state, and less than 50 MB for the standard deviation. In this case, since the assignment of values to keys is already somewhat balanced (due to the small skew values), the partitioning of the majority of the keys identified as hot-spots is not very meaningful to the overall storage load balancing. Despite that with parameter 0, RS detects more keys as hot-spots than with parameter 4 (since this is more restrictive), the resulting shards are small and do not make a relevant difference. In the end, with parameter 4, only the most popular (or bigger) keys—i.e., the ones that make the most difference—are partitioned.

As the skew value increases, we start to see an apparent difference in the maximum and standard deviation of the per group state for the configurations with sharding enabled. Additionally, the average per group state of all configurations starts to stabilize around the same values. With a skew of 0.75, Figure 5.20c starts to depict a big difference in storage load balancing. When sharding is disabled (i.e., OFF), the maximum per group state is 6 GB, and the standard deviation is around 350 MB. In turn, when sharding is enabled, the maximum per group state and the standard deviation remain almost unchanged from the previous skew values, at around 2 GB and 230 MB, respectively.

Naturally, when moving to more restrictive parameters, both metrics increase because less keys are considered hot-spots and partitioned. From all the hot-spot detectors, HTA is the one achieving the lowest values from all the others for these metrics, and it does so consistently across all skew values. This is because of a rationale similar to the one as RS in the previous paragraph. Despite parameter 0 identifying more keys as being hot-spots than parameter 4, since it looks at the average size per key in the local store, the difference in the number of partitioned keys is very small. Additionally, the keys that are partitioned are the ones that make the most difference. In turn, RS achieves the highest values from the four detectors for these two metrics. Here, the values start to make a turn, and grow as the parameter becomes more restrictive. With this skew, some keys start to become much more popular than others, and partitioning them even with less restrictive parameters makes a relevant difference in the overall result.

For skew values of 0.99 and 1.25, in Figures 5.20d and 5.20e, respectively, the plots are very similar, but with increasing values for both the maximum per group state and the standard deviation. From a skew value of 0.99 to 1.25, the popularity of the most popular key more than doubles (from 8% to 19%, or from 4 000 to 9 500 values assigned). For instance, with a skew of 1.25, the most popular key has almost 20% of all the values in the overlay, thus creating a big imbalance in storage load. This can be verified when the sharding mechanism is disabled (OFF in the plots), where both these two metrics also more than double from one skew value to the next. In this case, with a skew of 0.99, the maximum per group state is almost 20 GB and the standard deviation is around 800 MB. With a skew of 1.25, the maximum per group state is more than 48 GB, and the standard deviation is around 1.7 GB. However, with sharding enabled, all hot-spot detectors are able to greatly reduce this imbalance. Note that for these two skew values, the plots showing the maximum per group state (the left column) present the y axis with a logarithmic scale. With a skew of 0.99 and sharding enabled, the maximum per group state is greatly reduced to around 2–5 GB, and the standard deviation to around 220–360 MB (depending on the configurations). In turn, with a skew of 1.25 (and sharding enabled), the maximum is drastically reduced to around 2–6 GB, and the standard deviation to around 250–550 MB (depending on the configurations). With the less restrictive parameter (i.e., parameter 0), this results is an average decrease of 92% for the maximum per group state, and of 80% for the standard deviation. Whereas, for the most restrictive parameter (i.e., parameter 4), this still results is an average decrease of 82% for the maximum per group state, and of 62% for the standard deviation.

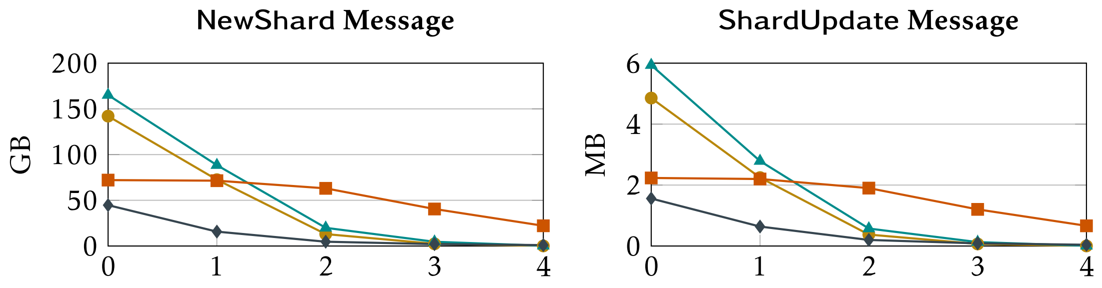
All hot-spot detectors, from medium to high skew values, start to have larger values for maximum per group state and standard deviation as parameters become more restrictive, because less keys are identified as hot-spots and thus partitioned. The exception is HTA, that is able to maintain its values independent from the parameters. Since this hot-spot detector bases its decision in the average size per key of the local store, keys are partitioned trying to reach a stabilization (average) value. Here, note that the vast majority of the keys that were considered hot-spot are due to the first part of the criteria (above 95%), i.e., due to the local size being above the average size per key, and not because there is only one key in the local store. Also, albeit marginally, HTA manages to consistently achieve a lower average than any other alternative across the majority of the skew values. In turn, RS presents the largest maximum per group state from all the detectors. This is due to the fact that, it bases its decision in a ratio from the local key store. For instance, parameter 4 considers keys to be hot-spots only if their local size represents more than 80% of the global size. As a consequence, only very large keys are going to be partitioned.

In the end, Figure 5.20 perfectly depicts the effects of skewed (or non-uniform) data on the storage load of an overlay. The storage load imbalance caused by this skewed data can greatly hamper the overall performance of an overlay. Some groups/peers will store much more data than other. Besides this representing unfair resource usage, which can be important when referring to less resourceful devices (e.g., mobile or other edge devices), it can also entail that these peers will receive and process much more read requests, while other are relatively idle. Hence, working against the fairness of both storage and query loads in the system. Thus, revealing the importance of such a sharding mechanism.

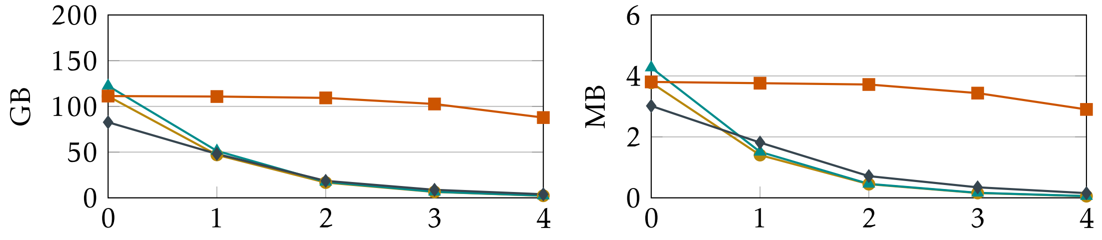
In a different aspect, Figure 5.21 illustrates the amount of transmitted messages (in GB) related with the sharding mechanism, namely NewShard and ShardUpdate messages. By looking at the figure, we can see that the plots for both messages follow the same trend, but with a major difference in the values' magnitude. This is because of the content of each message type. The NewShard messages carry the new opkey, the shard index, and the set of values assigned to the newly created shard (which represents the bulk of the message). In turn, ShardUpdate messages only carry the opkey, the new sid, and, in some cases, the oids of the values to be removed. Thus, NewShard messages are clearly the ones that have the bigger impact in the overall transferred data.

Overall, Figure 5.21 reveals the other side of the coin regarding Figure 5.20. Here, this sharding mechanism presents a natural trade-off between the achieved storage load balancing and the amount of transferred data. Partitioning a popular key into smaller portions means that some values assigned to that key have to be transferred to another group(s). Thus, the system configuration should be done in such a way that it does not entail an excessive amount of data transfers, balancing this trade-off.

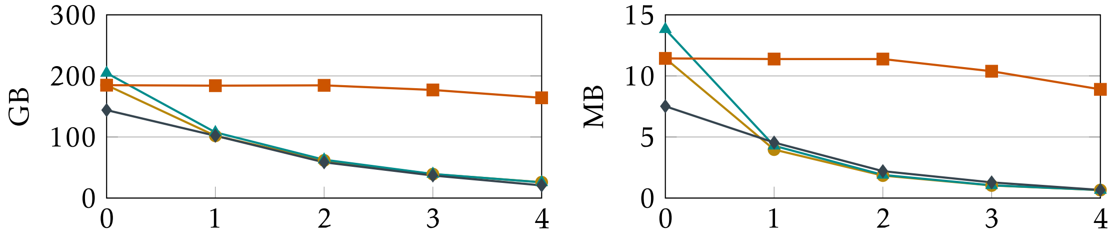
Figure 5.21a showcases clearly the problem of over-partitioning keys. With skew 0, parameter 0 for both AA and AS was deliberately set below the average number of five values per key. In this case, these hot-spot detectors identify keys as being hot-spots when one has more than three values or more than 15 MB assigned. Since we are practically



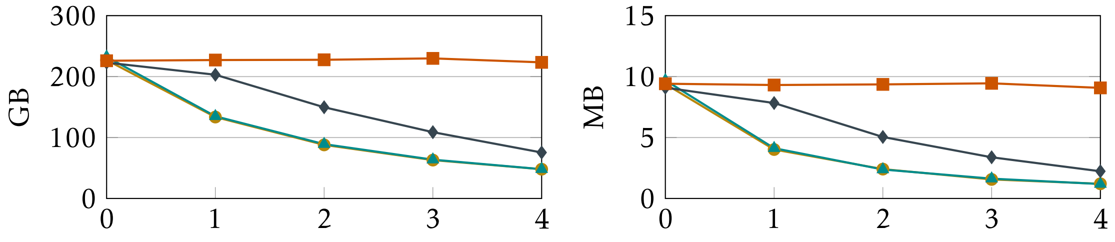
(a) Skew 0.



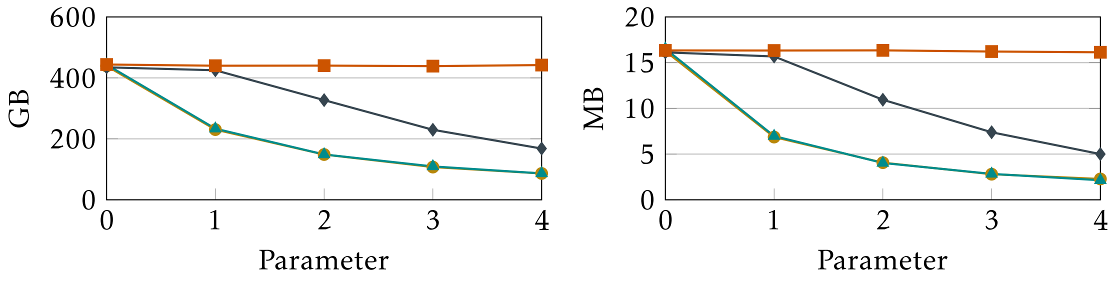
(b) Skew 0.5.



(c) Skew 0.75.



(d) Skew 0.99.



(e) Skew 1.25.

Figure 5.21: Transmitted messages related with the sharding mechanism in Parsley.

partitioning every key, the amount of exchanged data resulting from sharding-related messages is too high—requiring the transfer of around 150 GB of NewShard messages system-wide. As the parameter becomes more restrictive, the amount of transferred values drops quickly, since less keys are considered popular and partitioned, and thus less data needs to be transferred.

From Figure 5.20, we can see that for small skew values (namely 0 and 0.5), there is no meaningful difference between using sharding or no sharding at all. Thus, the amount of transferred data depicted in Figures 5.21a and 5.21b cannot be justifiable, considering the return they bring in terms of effective storage load balancing. In turn, for increasing skew values, specially the more restrictive parameters start to present reasonable trade-offs when comparing the achieved storage load balancing and the required amount of transferred data. For instance, in Figure 5.21e, with a skew of 1.25, we can see that with AA and AS for the more restrictive parameter are transferred around 86 GB in NewShard messages and roughly 2 MB in ShardUpdate messages. Looking at Figure 5.20e, this translates in a reduction of 90% for the maximum per group state and 70% for the standard deviation, which we claim is an acceptable trade-off. Moreover, more restrictive parameters (or other kinds of hot-spot detectors) can be studied in order to require even less data transfers.

The AA and AS hot-spot detectors react almost in the same way in every scenario. Data transfers drop quickly as the parameter increases for every skew value. Also, they provide the lowest data transfers from all the detectors. In the first three skew values, AA and AS require the most data transfers for the first parameters, and then, quickly drops to very small values. For the next skew values, these hot-spot detectors continue with this trend, but decreasing the amount of required data transfers in a more soft way. This is directly related with the configuration parameters used and the skewness of the data.

Similar to Figure 5.20, HTA requires a stable amount of data transfers when varying the parameter. It achieves the best results in terms of storage load balancing, at the expense of large data transfers. For instance, with a skew of 0.75, in Figure 5.21c, it requires the transfer of around 200 GB of NewShard messages and roughly 10 MB of ShardUpdate messages. In return it provides a reduction of almost 70% in the maximum per group state and of 43% in the standard deviation. HTA is always the hot-spot detector that requires the most data transfers in all scenarios (except when AA and AS over-partition keys, and have small parameters until a skew of 0.75).

Still, for small skew values, RS requires the least data transfers. As the skew value increases, AA and AS start to get closer to RS. With a skew value of 0.99, this scenario inverts, and RS begins to require more data transfers than AA and AS. However, in this case, AA and AS provide a better trade-off, because they offer better storage load balancing than RS, and at the same time require less data transfers. For example, in Figure 5.21d, with a skew of 0.99 and for parameter 4, AA and AS reduce the maximum per group state by 10% more than RS, and achieve this by requiring 50% less data transfers for both sharding-related messages.

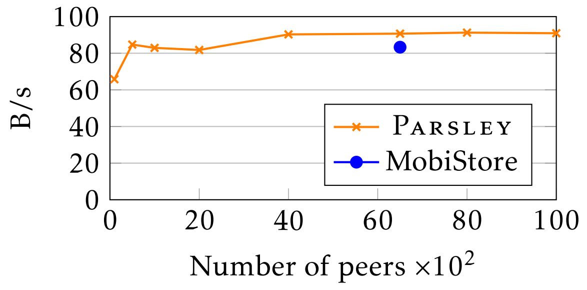


Figure 5.22: Per peer overlay management overhead.

5.5.4 Overlay Management Overhead

Lastly, we also decided to do a small experiment to gauge the overlay management overhead. This overhead concerns all the overlay maintenance procedures, namely, finger table updates, stabilization, group maintenance, and passive view gossip. For this, Figure 5.22 shows how the per peer overlay management overhead behaves as the network size grows. These results were measured with the overlay in a steady-state, after all peers entered the overlay and a stabilization period as passed.

As we can see, the management overhead grows very slowly with the network size. For small network sizes, the overhead grows slightly. In turn, for large network sizes, the overhead starts to grow really slowly, almost stabilizing. In fact, from a network size of 100 to 500 peers, the overhead grows around 22%. However, the average growth rate of the management overhead for all the measured network sizes is of roughly 4%, a value we claim to be completely acceptable.

Additionally, we also try to make a small comparison with MobiStore regarding this metric. According to the results presented in the MobiStore paper [139], in a network with 6 500 peers, it reports an average of around 83.3 B/s of per peer management overhead (the blue data point in Figure 5.22). In turn, Parsley in a network with the same size reports an average of around 90.7 B/s. Also, in the paper, the MobiStore authors imply that this specific experiment was done differently from the rest, using a combination of full and "diff" updates of the aggregated routing tables. Still, note that in the MobiStore paper, the authors are not clear as to what is included in this per peer management overhead metric (i.e., what types of messages are counted in). Hence, here we risk not making an apples to apples comparison.

In the majority of MobiStore experiments, peers leave the network at exponentially distributed intervals with a session time ranging from two minutes to one hour. The authors explain this is to mimic the short session times of mobile devices. After completing an "active" session, peers leave the network for periods ranging from zero to 20 minutes, thus, some peers end up to not actually leaving the network. In turn, in Parsley's churn evaluation, the amount of churn applied to the overlay is the actual number of peers removed from the network. Thus, in the end, the availability benefits and robust operation of Parsley, showcased by this evaluation section, are achieved at the expense of (possibly) some extra overhead—an estimated marginal increase of around 8% in management overhead when comparing with MobiStore. Also, to the best of our knowledge, no other related work system was experimented with such high churn scenarios.

5.6 Concluding Remarks

In this chapter, we tackle the dynamic population of cells/groups and the case of data hot-spots in structured overlays—two issues found in our previous Thyme-DCS approach (§4.5). Namely, we present Parsley, a flexible group-based DHT that provides robust and efficient data storage while enabling load balancing for both query and storage hot-spots. To achieve this, Parsley combines techniques from both unstructured and structured overlays. The unstructured component comes from peers being clustered into groups of flexible size, and work collaboratively within them to act as virtual peers in the structured layer. Groups are used to simplify data replication and enable query load balancing, while at the same time providing increased resilience to churn. We diverge from previous related work by employing a preemptive peer relocation technique, where individual peers are proactively relocated from larger into smaller groups, as a way to avoid merging (which requires costly state transfers). In turn, to tackle storage hot-spots, Parsley employs a dynamic data sharding mechanism, inspired by the multi-publication replication technique. When a group finds a popular key storing many values, it starts sharding the key, dynamically partitioning the mapped values among other group, in order to scatter the key's storage load.

Parsley is flexible because it allows the configuration of several of its decision criteria, providing a useful tool to study the impact of such strategies. For instance, it is parametric on the new peer acceptance logic or the hot-spot detector. Thus, contrary to DEB Tree, our solution for storage hot-spots allows a more dynamic (pluggable) configuration of the popularity criteria.

To evaluate Parsley, we implemented a prototype in PeerSim and used it to experimentally validate its mechanisms, through simulations of a large-scale system. Namely, we focused on assessing Parsley's resilience to churn and the benefits of the PPR technique, and the trade-offs of our dynamic data sharding mechanism. Our evaluation shows the big resilience of the overlay to churn and the effectiveness of the PPR technique in reducing the amount of required merge operations, and consequently the required bandwidth for data transfers. Additionally, it presents the trade-off imposed by the dynamic data sharding mechanism, between storage load balancing and the amount of transferred data (due to key sharding). The adequate configuration of some hot-spot detectors shows that they can provide a good balance between both metrics, yielding a positive storage load balancing for very popular keys at an acceptable cost. Overall, results show that Parsley does indeed incur in less merge operations, requiring smaller bandwidth costs;

achieves high availability in face of (intensive) churn; and promotes good storage load balancing in the presence of skewed data.

Just to illustrate the big benefit of the group-based approach, imagine the following scenario. In the perfect case (with some probability), it is even possible for Parsley to tolerate the simultaneous loss of 60% of the peers in an overlay with 10 000 peers (scattered among roughly 1200 groups), without requiring any topology change (i.e., without any data loss and without requiring any state transfer among groups).

5.6.1 Discussion

Regarding the PPR mechanism, the evaluation shows its effectiveness in reducing the amount of merge operations, and consequently the required bandwidth for data transfers. In situations without peers entering the overlay, Parsley yields a savings of around 15% in data transfers due to merge operations. In turn, with peers also entering the overlay, PPR yields a savings of up to 40% in data transfers. Despite the studied alternatives (Push, Pull, and FPPR) having slightly different results, the overall savings are very similar. The only difference worth mentioning is that the Push alternative ended up requiring less peer relocations to achieve almost the same savings in merge operations, when comparing with FPPR. However, regarding data transfers, the difference between the two is negligible.

Regarding the dynamic data sharding mechanism, there are several considerations to make. All alternatives with sharding enabled have showed very generous results concerning the reduction of the per group state. With a high skew value of 1.25, the hot-spot detectors achieved a reduction of 82–92% for the maximum per group state, and a reduction of 62–80% for the standard deviation (depending on the configurations) when comparing with the system without sharding. On the other hand, the results for transmitted messages related with the sharding mechanism show the trade-off brought by this mechanism. The reduction of the per group state has a direct relation with the amount of data transferred by this mechanism. Thus, this calls for a careful configuration of the sharding mechanism, and most importantly, of the chosen hot-spot detector. Despite the hot-spot detector logic could be injected into Parsley, we provide four different implementations out-of-the-box: AA, AS, RS, and HTA. Since they use absolute values, AA and AS seem to be better for scenarios where the object/value size does not vary much and is know a priori. In turn, HTA achieves the best storage load balancing but at a high cost, requiring massive data transfers. Lastly, RS shows to be a good choice for high skew values. Namely, with more restrictive parameters, it offers a good trade-off between load balancing and required data transfers.

In the end, this sharding mechanism presents an evident overhead of having to transfer data as keys are partitioned. However, if the shards of the most popular/relevant keys are already in place, and peers know about them, when put operations are issued, they will be scattered among the existing shards. Thus, effectively balancing the storage load in the overlay, and working to reduce the required data transfers (i.e., the NewShard messages). Therefore, this insight may show evidence that some kind of (background) shard dissemination mechanism among the overlay peers could be beneficial.

5.6.2 Future Work

Currently, we are implementing Parsley's approach in a fully-fledged system, in order to be able to evaluate its behavior in real-world scenarios. More specifically, we are integrating Parsley as the node clustering process in Thyme GardenBed (see §6).

Additionally, as future work, we highlight the following directions: an integrated approach to load balancing, taking into account the amount of incoming requests (per time window) to groups and the specific resources available in each group's peers; partial replication inside groups, as another way to tackle storage hot-spots; make the split logic to consider peers' resources when dividing a group; implement a resource-aware leader election mechanism, favoring more resourceful peers; and study more dynamic scenarios with workloads where the data skewness varies over time.

Another possible direction is the study of other (more intricate) hot-spot detectors. For instance, trying to use other data indicators, or even machine learning, to understand which keys are critical. Thus, focusing in the sharding of those keys, as a way of reducing the data transfers to the strictly necessary.

Chapter 6

Data Storage and Dissemination in Multi-region Edge Networks

"Let everything happen to you. Beauty and terror. Just keep going. No feeling is final."
— *Rainer Maria Rilke*

This chapter reports about data storage and dissemination for co-located mobile devices with access to network infrastructures. More specifically, it reports about Thyme GardenBed, a framework for data storage and dissemination across multi-region edge networks, that makes use of both device-to-device (D2D) and edge interactions. In this work, we address the two lower levels of the network hierarchy and their symbiotic integration, i.e., the end-user devices and the edge (servers).

We start by introducing some context and motivation in §6.1. Next, we present a review of some related work and a comparison with our approach in §6.2. In §6.3, we discuss the design decisions behind GardenBed, the framework connecting edge regions. After, in §6.4, we detail the whole system, Thyme GardenBed. Then, in §6.5, we present the architecture of GardenBed's caches. Afterwards, we report the experimental evaluation of Thyme GardenBed in §6.6. We conclude the chapter in §6.7, presenting our conclusions, some future work, and the publications resulting from this work.

6.1 Introduction

Mobile devices have become *pervasive* in today's society, being an essential tool for people to communicate and perform daily activities or even work-related tasks. Along with the soaring introduction of mobile-tailored applications and services, as well as the paradigm shift of working on-the-go, a recent study predicts global traffic generated by non-stationary devices will grow 600%, from the seven monthly exabytes recorded in 2016 to 49 exabytes per month in 2021 [56]. This study also confirms that mobile clients will increasingly *offload* their interactions and data requests to nearby infrastructure stations, such as public Wi-Fi hot-spots, instead of cellular networks. Accordingly, the prediction is that Wi-Fi hot-spot deployment will grow six-fold to 541.6 million in 2021. However, although the hardware capabilities of these hot-spots have greatly increased, mobile communications still remain a bottleneck for most applications. This is partially caused by the use of cloud infrastructures, which effectively represent a large-scale *communication hub* where many platforms compete for processing power and channel throughput [110].

The *edge computing* paradigm takes advantage of the resources available at the network edge to bring cloud services closer to end-users. The key insight is that it is more efficient to communicate and distribute information among *nearby* devices than to use distant centralized intermediaries [117]. By processing and storing data near its source, applications can be more responsive while *relieving* some of the load from cloud and network infrastructures, potentially also providing increased data privacy and ownership.

Several proposals leverage edge servers to bring (or cache) remote content closer to the mobile devices [77, 282, 292]. Others, promote D2D communication among nearby devices to reduce (or eliminate) traffic to cloud services [79, 243, 249]. Our work relates more closely to the latter. We tackle the challenge of sharing persistent data *generated by* mobile devices (e.g., social media, or automatically collected sensory data) among users in other networks, such as work colleagues in another building or people with similar interests in a large venue (e.g., a football stadium). The novelty comes from the fact of doing so *without* requiring data to persist on edge or centralized services, i.e., data can reside at its source—in the mobile devices—and be shared through D2D interactions. We also make a novel use of the edge servers, which do not serve as caches for data stored in cloud infrastructures, but rather as caches for data *generated* and *available* in the mobile devices. Hence, we *cooperatively* integrate and leverage *both* edge servers and mobile devices to take advantage of a wider range of resources available at the network edge, and sharing the load between both.

In this chapter, we propose THYME GARDENBED, a framework that leverages both D2D interactions and edge servers for efficiently storing and sharing content in a network of networks of mobile devices that spans across several edge network regions, such as Wi-Fi basic service sets. D2D communication is used for intra-region interactions, such as obtaining locally available content, whilst edge server communication is used both to: a) cache data generated by the mobile devices, so that this data may also be obtained from the infrastructure, and with that *move load* from the battery-constrained mobile devices to the servers; and b) make mobile devices aware of content generated in other regions, and provide the means for such content to be retrieved. Figure 6.1 depicts a scenario where spectators of a sporting event may share photos or videos of the game.

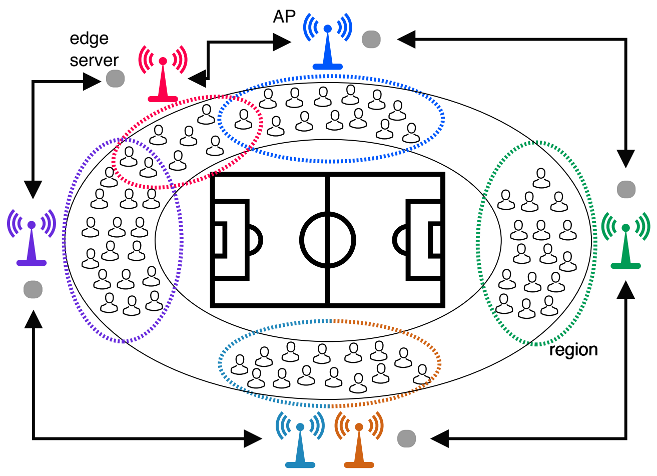


Figure 6.1: Thyme GardenBed example usage scenario in a football stadium [278].

Data sharing within a region may be performed via D2D or by communicating with the local edge server. In turn, data sharing across regions requires disseminating such data to the edge servers, announcing its availability to the interested users, and then transferring it from the edge to the mobile devices.

In summary, the main contributions of the work presented in this chapter are the following: 1) the Thyme GardenBed framework, which is, to the best of our knowledge, the first content storage and sharing system that makes use of both D2D and edge interactions across multi-region edge networks; and 2) the framework's extensive experimental evaluation, both in real world and simulation scenarios—the evaluation includes the comparison of our approach to one using a cloud infrastructure, and to another using only mobile devices.

6.2 Related Work

Works such as [175, 191, 245, 290] have dealt with content dissemination in ad-hoc and delay tolerant networks. Although these works share similarities with Thyme GardenBed, our focus is more on data storage and dissemination between nearby devices when these are connected to an infrastructure (possibly via multiple access points (APs)). In this field, we may divide the existing approaches in two categories: systems that use mobile devices to store/cache data [249, 279, 282–284]; and systems that use edge infrastructure nodes to store/cache data [9, 273, 292].

Regarding the first category, in [283], the authors define a D2D replication strategy using information obtained via social graphs, and leverage device mobility to propagate data among different regions. The proposed solution allows mobile devices, in multiple regions/APs, to cache data items (from cloud services) based on content popularity and user mobility patterns. Then, devices can obtain data directly from the content servers, or through D2D communication, by contacting other devices in range.

In [279], authors exploit user mobility and propose a mobility-aware caching placement strategy to maximize traffic offloading from the cellular network into D2D links. By caching popular content at devices, mobile users may acquire required files from close by user devices via D2D communication, rather than through the cellular network.

PDS [249] focus on data discovery and retrieval on networks of co-located devices. It is inspired in information-centric networking and mobile devices employ an aggressive caching policy. Since data is only cached if requested, less popular data may disappear.

In MECCAS [284], the edge network is composed of request and/or storage nodes, where the former are devices that ask the system to store data and the latter being devices that lend storage resources to the system. It leverages the hardware information of neighboring devices to dynamically decide which of the storage nodes will be assigned to store content.

PopPub [282] is a distributed collaborative caching strategy, also inspired in information-centric networking, where nodes leverage the popularity aspects of each data item to make caching decisions.

In turn, Thyme GardenBed is able to make caching (and pre-fetching) decisions using only local information gathered by (and from) the mobile devices in a region. It makes cautious decisions on what to cache based on a user-defined ranking algorithm taking into account limited storage. It also employs a functionally symmetric architecture, where all mobile devices contribute fairly with resources to the system. Furthermore, it leverages the resources of edge servers to offload some of the storage and management responsibilities from the more resource-constrained mobile devices.

Contrary to the previous approaches, EdgeBuffer [292] takes advantage of the storage available at APs to bring content closer to mobile devices, and derives a prediction model based on network-level statistics. It employs a caching scheme that captures both long-term aggregated content access patterns and short-term individual user access patterns. Content prefetching for individual devices is achieved by predicting the next AP a device is moving to, and at what time. However, this approach leverages on the MobilityFirst architecture [213], where mobile clients' network association is logged by the network itself, thus facilitating mobility prediction.

SDMEC [9] proposes a cooperative mobile edge computing (MEC) storage architecture. It allows any user device inside the radio coverage of a given MEC node to store data. If the local MEC node has not enough resources, it stores all possible data blocks locally, and the remaining are transmitted to other neighboring MEC nodes, searching from closest to farthest, until all data is stored. Thus, different blocks of the same file can be scattered across a considerable geographical area.

CoPro-CoCache [273] is a framework that allows edge infrastructure nodes to collaborate on video caching and processing. Multiple edge servers work collaboratively to cache popular videos in different qualities, thus allowing clients to adapt their requests, e.g., to network conditions or devices' characteristics.

Contrasting with all these proposals, Thyme GardenBed capitalizes on the storage and computing resources from both mobile devices and edge servers, using the latter to connect multiple network regions, thus allowing the flow of (configurable) relevant content among regions. By supporting a user-defined ranking algorithm, which is fed with local application-level information, it provides flexibility, and lightweight statistics gathering. Also, we specifically address the caching of content generated by mobile devices. However, Thyme GardenBed is general enough and allows the caching of content generated from *any* entity from the upper network layers.

Additionally, there are some works following an identical approach to Thyme GardenBed. In [218], the authors present a case study application for video dissemination using a hybrid edge cloud architecture, featuring Android devices (possibly connected through WiFi-Direct) and cloudlets (connected in a WiFi mesh). This system is for scenarios with no network infrastructure access. Data is generated by the mobile devices and introduced into the system. Then, cloudlets work as simple caches, continuously synchronizing their cached video contents among them. In the end, all nodes (user devices and cloudlets) work to cache data and allow its sharing. Contrary to Thyme GardenBed, here, cloudlets have little intelligence, blindlessly caching videos. Similarly to the previous work, Ramble [95], is a system for opportunistic dissemination of content in environments with limited network infrastructure access. Mobile devices can share data among them through WiFi-Direct groups, and can connect to cloudlets when in range. In turn, both devices and cloudlets can connect to the cloud (when available) to increase the chances of data sharing. Another system [247], uses a centralized server for video dissemination, allowing mobile devices to interact directly (through Wi-Fi Direct) to share their data and offload some of the server's data sharing job.

There exist still other works that address the broad field of mobile edge caching [281], e.g., for connected cars [164, 293], incentive mechanisms [163], etc. However, Thyme GardenBed is not just a caching mechanism. It also brings some intelligence to the edge, by interconnecting edge servers and allowing data to flow among different edge network regions. Furthermore, to the best of our knowledge, few systems use cooperatively the resources of both the mobile devices and the edge servers like we do. We also highlight that, as far as we know, the majority of these works only provide analytical studies and/or simulations, whereas we analyze our proposal in both simulation and real world experiments, providing better insights on how our system should behave in realistic scenarios.

6.3 The GardenBed Concept

Such like Yggdrasil[1], the idea for GardenBed emerged from the desire to interconnect several individual edge network regions (e.g., APs), in order to aggregate data from different clusters of devices (possibly in the same venue) into one cohesive and consistent

[1]In Norse mythology, it is the tree at the center of the cosmos, connecting all nine worlds.

end-to-end storage network. In such scenarios, data generated by devices in the same venue, although that connected to different APs, might be of interest to many other devices scattered throughout the venue. Thus, when connecting those regions together, we also call for a way that data would be able to flow between them. Additionally, in order for this interconnection to be beneficial to all involved parties, it needs to favor reduced energy costs and communication latency (mostly from the mobile devices' perspective).

In line with the requirements described above, GardenBed was born, and integrated with Thyme, forming an ensemble of several plantations of Thyme. In order to make GardenBed more general, we devised a corresponding client middleware component, GardenBed Client, abstracting from the data storage system it attaches to. Thus, GardenBed aggregates and connects networks of any kind of edge storage system that attaches itself to a GardenBed Client.

Since Thyme works exclusively in the lower level of the network hierarchy (i.e., the end-user devices), the only way forward was to go up the hierarchy, searching for new and different resources. Thus, in this case, it meant going to the second level of the network hierarchy—the edge—, and exploit the resources available there. Accordingly, as presented in Figure 6.2, GardenBed is the component running on the edge servers, aggregating the several instances of Thyme running in the network.

With all this in mind, the major requirements already identified are: wide-area data sharing, reduced communication latency, reduced energy costs, and flexible configuration (of some components).

Regarding wide-area data sharing, edge servers work in multiple ways. First, by being connected among each other and exchanging periodic requests and replies for data, it allows for the flow of data between regions. By being a periodic mechanism, it allows communication batching which enables better bandwidth usage. Next, by using them as a cache for data from different regions, it allows them to also work as another replica in the system (with possibly added guarantees), increasing data availability and bringing data closer to the devices. Also, data exchanged between regions and stored in cache is filtered by a programmer-defined ranking algorithm. This ranking algorithm allows the definition of what data is relevant in the system, enabling data to flow among regions, while at the same time reducing the communication overhead to what is deemed necessary. All this enables the sharing of data management among the mobile devices and the edge servers, and places some intelligence at the edge, spending resources judiciously.

Regarding reduced latency and energy costs, on the one hand, since edge servers are close to the mobile devices (preferably attached to APs), they are able to serve data with reduced latency when compared to typical cloud solutions. In fact, even sharing data among edge network regions can provide lower latencies than accessing the cloud. On the other hand, since mobile devices offload some of the system management to the edge servers, it works as an energy shift, by moving some of the energy expenditure from the mobile devices to the edge servers. In the end, this works both to provide access to data with low latency, and at the same time share the system workload among the entities in the two lower levels of the network hierarchy.

Regarding flexible configuration, edge servers do not have any specific requirements. They just need to have some computing and storage resources. In turn, GardenBed (i.e., the software component) has several configurable components (see §6.4.2), allowing the proper fine-grained adjustment of the system configuration to the resources available at each edge server. Nonetheless, edge servers should preferably be connected directly to the AP of the region they are managing (through a wired connection).

In the end, GardenBed promotes a symbiotic collaboration in three levels: among devices through D2D interactions, among edge servers, and between devices and their corresponding edge (region) server. From this, resulting an outcome beneficial to all actors: the Thyme GardenBed ensemble.

The work in this chapter was developed in the context of the M.Sc. of Vieira [278], more specifically the GardenBed concept was instantiated in Thyme (with its corresponding implementation and evaluation). My direct contributions in this work were mostly on the design of the solution and its architecture (and less in its implementation and evaluation). Following the more precise contributor role taxonomy (CRediT) [10], I contributed in the following ways to the work reported in this chapter: conceptualization, software, writing - original draft, writing - review & editing, and supervision.

6.4 The Thyme GardenBed Ensemble

Thyme GardenBed is a distributed system comprising a set of *stationary nodes* that execute at the edge of the wired network, and a set of physically distributed *mobile nodes*.

Stationary edge nodes (that we will refer to as *servers*) have a *1-to-N* relationship with the wireless APs responsible for managing the regions, as depicted in Fig. 6.1. The servers can run on the APs (if they have the necessary resources) or on simple micro-computers attached to the APs (like cloudlets). Furthermore, servers may also communicate with cloud services or external databases if desired, for instance for archival or analytics purposes. In turn, each mobile node (or *client*) belongs to a single region (although it may be in range of multiple APs), or is able to communicate via multi-hop communication to a node in a region. We do not impose any mobility restrictions, other than those imposed by the venue they are in, and the natural speed limits of humans. For communication optimization purposes, mobile nodes may be clustered in groups (e.g., Wi-Fi Direct groups [265]), as long as some can communicate directly with the region's server.

The servers run the GardenBed server component to collectively form a distributed system at the edge that offers a cross-region time-aware reactive storage (TARS) abstraction (§3). In turn, mobile devices run a system that enables content storage and sharing among devices within a region. In this work, we build on top of Thyme (§4), a data storage and dissemination system implementing TARS in wireless edge environments. The resulting ensemble, Thyme GardenBed, enables mobile devices to make data they generate available to peers in multiple regions.

The published data remains (initially) on the devices themselves (with an option of replication among region peers, for data persistence), but becomes automatically accessible to others in the same region. The role of the servers is to collect and cache data published in their region to: 1) make it available to other servers and, subsequently, to other regions; and 2) to share the burden of data dissemination in a region with the mobiles nodes. As a result, mobile nodes are able to retrieve data generated in any of the connected regions.

On the content-generation side, crossing region boundary is not granted. To limit the inter-region traffic to what is relevant (and to enable the definition of the meaning of "relevant data"), region boundary crossing can be subjected to some filtering, coded in a programmer-defined ranking algorithm.

The system presents a TARS interface, providing the following operations to users: insert, delete and retrieve data, and subscribe and unsubscribe to queries (through tags associated with the data objects). Data persistence allows for publications to be visible to subscriptions issued for the past, as well as for the future (i.e., subscriptions have a time scope). As a result, nodes joining the system later, or simply facing momentary disconnection, may still retrieve data published when they were absent. The inclusion of the delete operation permits the deletion of a previously published data item, so that it is no longer visible to subscribers. Notifications do not carry data, but only *metadata* that includes the data's location(s). The effective data item retrieval is triggered through the retrieve operation.

Even more, the data objects' tags are bound to *namespaces*, which allow for the coexistence of multiple applications, or different namespaces within an application.

6.4.1 The Mobile Clients

Thyme (§4), in its Thyme-DCS approach, follows a data-centric storage concept, using a key-value substrate built on top of a cluster-based distributed hash table (DHT). Nodes are clustered into *cells*, being that messages addressed to a cell are delivered to all nodes within. The use of the cluster-based DHT is two-fold: 1) cells are used to store all the system data; and 2) cells are exploited to match subscriptions against published content, i.e., acting as virtual publish/subscribe (P/S) brokers. In §4.5, the cluster-based DHT takes the form of a geographic hash table, enabling Thyme to be used in ad-hoc mode, where nodes communicate via D2D. In this work, we are more interested in using Thyme on an *infrastructure-based* setup, where nodes are connected to APs. Thus, the *node clustering process* does not depend on proximity relations but rather on the number of nodes and the amount of data stored in a region. This clustering process is a research topic on its own and falls outside the scope of this chapter.

In Thyme, content has associated metadata comprised by, among others, a small *description* and a set of *tags* related with the content. Metadata is indexed by all its tags, i.e., the cells resultant from hashing each tag replicate the metadata. If desired, the actual

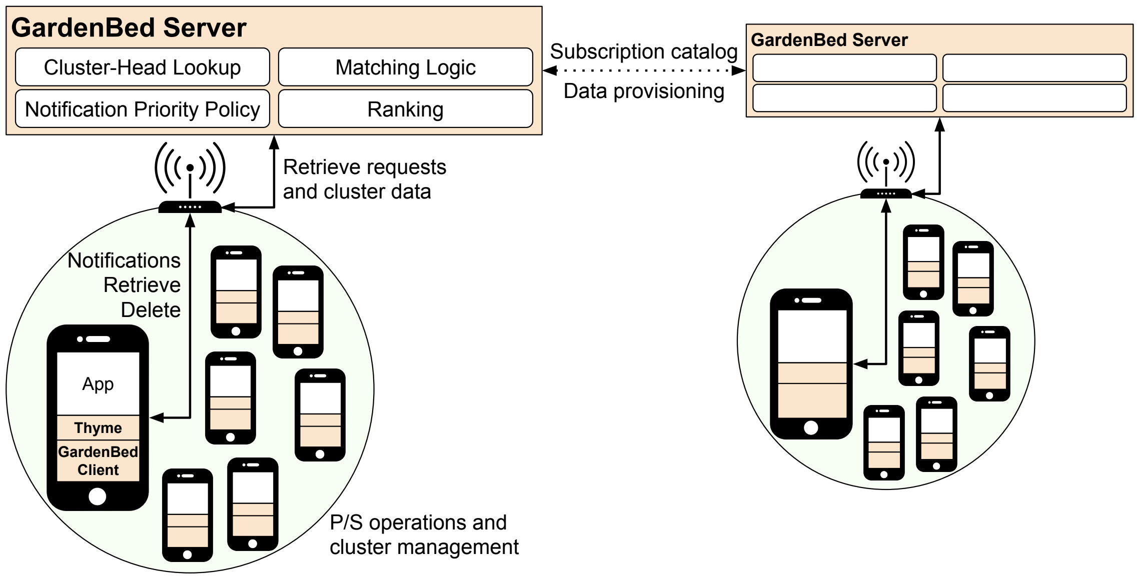


Figure 6.2: Architecture of Thyme GardenBed.

content may also be replicated in all the nodes of the publisher's cell (at the moment of the publication). A subscription comprises, among others, the *query* defining which tags are relevant, and the *timestamps* defining when the subscription's time frame starts and expires. Hashing each of the query's tags determines the cells where to send the subscription. These cells become (virtual) brokers for the subscription, and are responsible for checking if stored content matches the subscription, notifying the subscribers, if need be. By inspecting the item's description (such as a photo thumbnail), a notified subscriber may then decide to retrieve the item, from the list of received locations, or not.

6.4.2 The Edge Servers

GardenBed is a generic and flexible framework that requires the concrete implementation of several modules on the server side, as depicted in Figure 6.2. The *Cluster-Head Lookup* algorithm determines which is the cluster node to contact when performing operations over a given data item. The *Matching Logic* algorithm matches publications with subscriptions. The *Ranking* algorithm determines which data items must be uploaded to the edge. The *Notification Priority Policy* algorithm determines whether a notification should be sent by the server, by the mobile nodes, or by both.

As already mentioned in §6.3, edge servers do not have any special hardware requirements (or regarding any other resources). Nonetheless, the software component running on them should be configured taking into account the available resources (and any other software potentially running there).

Communication between mobile nodes (i.e., clients) and their server is bidirectional, being that most of it may be directed to the cluster-heads. Client → server communication informs the latter of the operations performed within the region, and enables the retrieval of data only available in other regions. The local information, named *cluster data*, is sent at cluster-level by the cluster-head, and includes the cluster nodes' subscriptions, delete operations issued since the last communication round, and statistical data on the downloads done for each data item—a metric that may be used to rank items according to their popularity.

In turn, server → client communication serves four purposes: 1) notify clients about fresh data, published in other regions; 2) notify clients of changes in the server, so that clients may update the metadata of a given data item to comply with those changes (e.g., the item has been evicted from the server's caches); 3) retrieve a data item, to be cached in the server, or to serve a retrieve operation from a client in a remote region; and 4) delete a data item, so that the client triggers its removal from the system.

As in Thyme, both in the clients and in the servers, we use a plain in-memory hash table as the native object store (in our case, a concurrent hash map in Java). However, any other storage engine could be used (e.g., (in-memory) databases).

6.4.3 Intra-Region Publish/Subscribe

The insert operation is purely local to a region, and remains unchanged. Data is kept on the device that issued the operation, being only replicated among the device's cluster members, if active replication is selected (possibly on a per-operation basis). On the other hand, the associated metadata is stored in the cells resulting from hashing each of the tags associated with the item (§4.5.1).

Asynchronously to this process, the edge server collects and caches the most popular items within the region, according to the injected ranking algorithm. The items are stored in the *local popularity cache (LPC)* (see §6.5), allowing local clients to obtain the cached content directly from the edge, as an alternative to inquiring their neighbors; and serving the subscription needs of clients from other regions. Here, the motivation is to divide the processing of retrieve operations among clients and servers, so that mobile nodes may have their battery lifetime extended as much as possible.

In order to make an informed decision about which data items to cache, servers use the statistics stored in the cluster data. After collecting a significant amount of such data, the server periodically applies the *ranking algorithm* to choose the most popular items. Then, it sends a retrieve request for the associated content (data and metadata) to the mobile clients, and stores this content in the LPC. From this moment onward, those data items will be registered as being *on the edge* by the server, and this information is also communicated to the head of the cluster responsible for managing the items' metadata. It is then up to the cluster-head to disseminate this information within the cluster. The periodic nature of this workflow will continuously update the LPC to follow the needs and trends of the mobile clients in the region.

The server also matches every incoming subscribe operation against all the data available within its caches. To that end, it applies the installed *matching logic*. Each successful match is followed by a notification to the subscribing clients, if any.

Considering the fact that subscribe operations are transmitted to both the mobile clients (§4.5.4) and the server, there is a high probability of receiving notifications from both ends. Therefore, in order to avoid the transmission of duplicate data, Thyme GardenBed prioritizes the emission of notifications according to the pre-determined programmer-defined *notification priority policy* algorithm. To be effective, this decision must naturally be met by the clients, which must adapt their behavior to the policy defined for the server.

The incoming subscribe operations are also used to compute the region's *subscription catalog*, fundamental for the inter-region global P/S process (see §6.4.4). This catalog is periodically computed from scratch, so that the server does not have to keep state information about individual clients.

6.4.4 Inter-Region Publish/Subscribe

To connect multiple groups of clients and their data into one cohesive and consistent end-to-end storage network, Thyme GardenBed implements the notion of a *global* (i.e., cross-region) P/S process. This has the goal of allowing mobile clients within each region to access content published locally, but also relevant data published by other users in distinct venue locations, i.e., other edge network regions.

This process is composed of three main phases depicted in Figure 6.3, and uses two other caches (see §6.5). The *prefetch cache (PreC)* stores data and metadata entries for content originated from remote clients. It is periodically fed with data received from the rest of the system. In turn, the *other regions cache (ORC)* stores entries that were initially in the PreC and were later considered relevant (i.e., effectively retrieved) by at least one of the region's mobile clients. The content of the PreC is expected to have a considerable turnover rate. Hence, moving content deemed relevant to the ORC provides more persistence guarantees for later access and discovery from other clients.

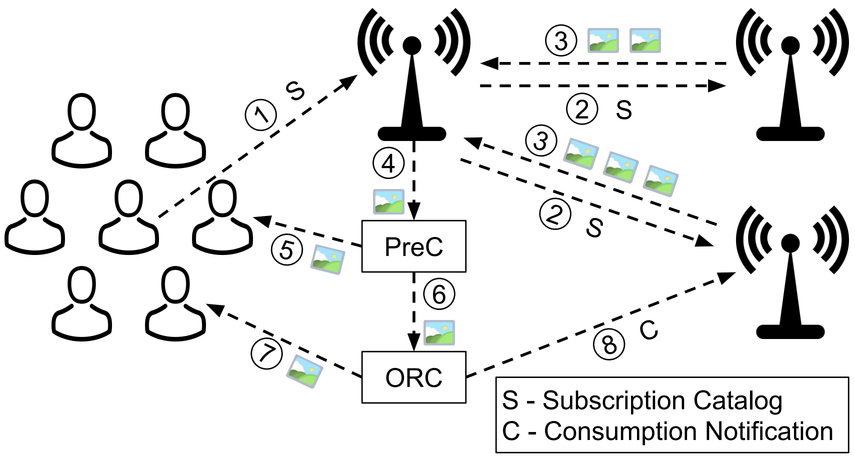


Figure 6.3: Global publish/subscribe execution process in Thyme GardenBed [278].

6.4.4.1 Dissemination of Subscription Catalog

Each server periodically disseminates its clients' *subscription catalog* to all the other servers (step ②). The message contains the server's PreC size, the number of subscriptions done by the region's clients up until the moment of transmission, and their actual subscription catalog by namespace, containing the subscriptions for each tag. This information is computed from the continuous flow of incoming subscribe operations (step ①).

6.4.4.2 Provisioning the Subscription Catalog

As already mentioned, inter-server data dissemination and provisioning is a periodic process. As a response to the subscription catalog previously disseminated by its peers (i.e., the other servers), a server looks up its LPC for content relevant to the needs of each server (according to the tags specified in the catalog) and sends, in batch, all of the matching data and metadata entries found (step ③).

By making use of the LPC to provision content based on the servers' subscriptions, we achieve a much better network utilization and performance, since we guarantee that only the most relevant (i.e. popular) data is transmitted. Also, the node's lookup process is directly executed in-memory instead of having to request the items on-demand from the mobile clients, and having a direct impact on the latency of the entire process.

The maximum size of each of the server's caches is configurable, and thus possibly distinct. This can potentially lead to cases where server *A* sends much more data than destination server *B* can hold in its PreC, resulting in an over-utilization of the network. Therefore, we capitalize on the received size of the server's PreC and trim the data to be sent, by removing exceeding entries, in order to meet the destination's cache size demands. Another optimization we employ during the provisioning is to use the subscription count for each tag, which effectively represents the subscription popularity (or relevance) for each tag, and calculate the appropriate size within the response to hold the relevant entries found. Thus, popular subscription tags will have more space in the message, while the opposite end will have less entries or even none.

Data Filtering. Not all content stored in the LPC may be of interest for a given subscription catalog. For instance, Thyme subscriptions are bound to time intervals, and thus only the publications abiding to the required intervals are useful. Here, the *matching logic* is once again applied to filter only the relevant items. The same happens in the *Provisioning the Subscription Catalog* phase, when looking up the LPC for data to supply to another region: the *matching logic* is applied to filter out obsolete entries (e.g., have publish timestamps outside of the intended time intervals).

6.4.4.3 Notification of Remote Publications

Upon the reception of a periodic *data provisioning message*, a server notifies the corresponding subscribers of the data's arrival, and stores the received items in its PreC (step ④).

These will remain there until they are retrieved and moved to the ORC (steps ⑤ and ⑥), or evicted by the arrival of new data.

The turnover rate of the contents of the PreC is expected to be quite high, due to the intrinsic nature of the *global P/S process*, which will be more accentuated as the number of edge servers grows. Thus, moving the consumed items to the ORC allows them to be stored in a more persistent manner and be easily accessed by other clients. Further requests to the same items from other clients will result in obtaining the data items directly from the ORC (step ⑦). 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 items to also be considered relevant by other clients, since it is expected for geographically-adjacent users to share similar interests to some degree.

6.4.5 Retrieving Data

Figure 6.4 depicts a simplified workflow executed for this operation. When a mobile client decides to retrieve a data item, it sends a retrieve operation to its server. It issues a *local retrieval* or *remote retrieval* for, respectively, content replicated in the current region or not. This information is present in the metadata.

For *local retrievals*, if the local server finds the desired data in any of its caches, it promptly serves the request, otherwise forwards it to the mobile replicas within the region. Regarding *remote retrievals*, the server performs a lookup for the specified item in the PreC and the ORC. If the item is not present, the request is rerouted to be processed in the source region, which first inspects its own caches and, if the lookup fails, retrieves the data from the mobile replicas. Recall that, if the data item was found in the PreC, it is moved to the ORC. Additionally, the server notifies the source peer of the fact that the item was effectively consumed by the current region's clients, in what we call the *Consumption Notification Optimization* (step ⑧ in Figure 6.3). By doing this, the source

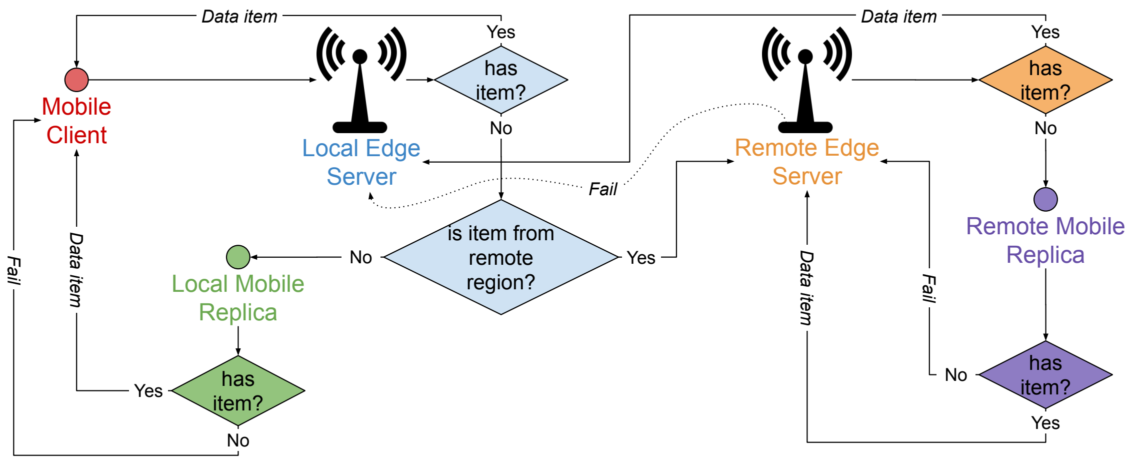


Figure 6.4: Workflow of the retrieve operation triggered by a mobile client (red circle) in Thyme GardenBed. The colors of the decision flows determine who executes them.

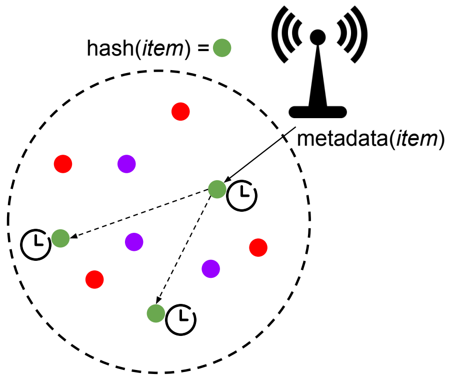


Figure 6.5: Local indexing of a remote data item in Thyme GardenBed.

server records this information and does not send the specified item again in the next *n Provisioning the Subscription Catalog* phases, saving network bandwidth (since the item is already in that server's cache).

When a server receives the reply to a *remote retrieval* with data from another region, before routing the requested item to the client, it proactively caches the incoming data in its ORC (since it is considered relevant). Thus, consequent requests to the same item, by users in the same region, will be served directly by the nearest server's cache.

Also, when a mobile client or a server tries to retrieve an item from a mobile replica, instead of asking just to one of the replicas, the actual implementation of this workflow takes into account all the possible mobile replicas for that item. Thus, it iteratively asks a subset of those replicas (during a retry procedure), until the data is successfully retrieved.

6.4.6 Local Indexing of Remote Data

The first client to retrieve a data item inserted in another region will cause the item to move from the server's PreC to the ORC, so that it may be available to other clients in the region (steps ⑤ and ⑥ in Figure 6.3).

Furthermore, we also allow the indexing of the given item by the mobile clients, as portrayed in Figure 6.5, so that it may be served via D2D. The time-to-live (TTL) of these items may be set in the system's configuration file. If the item is later evicted from the server's cache, the correspondent cluster(-head) will be notified to update the item's metadata. From that moment onward, the item will only be available via D2D, until its TTL expires. However, a new increase in popularity may cause it to be, once again, available from the edge and re-indexed in the region.

6.4.7 Deleting Data

The `delete` operation has a global scope, since the item to remove may have crossed the boundaries of the current region (§6.4.4), and thus may be cached in several servers, and even be indexed in multiple regions. Hence, as illustrated in Figure 6.6, whenever a server

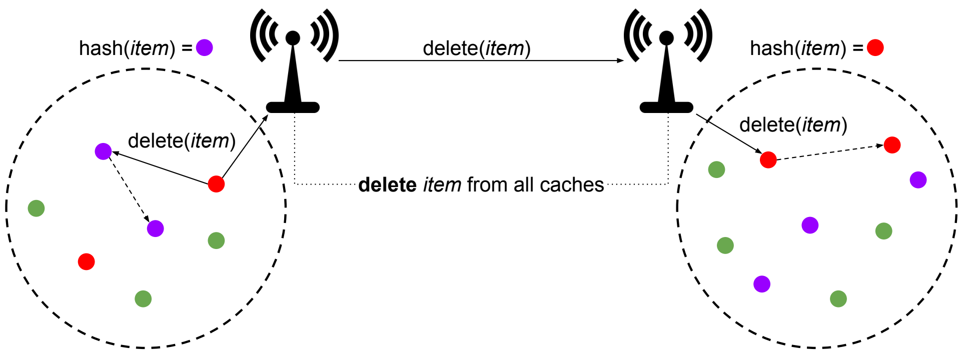


Figure 6.6: Delete operation in Thyme GardenBed [278].

receives a delete operation, it removes the entries associated with the referenced item in all of its caches (if present).

Moreover, except for the server of the region emitting the operation, they also propagate the operation to the cluster-head responsible for handling the item's metadata, for it to execute the operation locally. This workflow effectively guarantees that an item is indeed persistently removed from the system.

6.4.8 Election and Role of the Cluster-Head

Every cluster (or cell) has a cluster-head responsible for interacting with the server, to transmit the *cluster data* and receive communication directed to the cluster, i.e., to all cluster members. This approach trades-off the over-utilization of a single node's resources within each cluster for the resources needed to proactively keep cluster membership in the server, which would otherwise be necessary to reduce the overhead of failed communication attempts with clients that are offline or already left the system.

To avoid strict and computationally-heavy coordination between all nodes of a given cluster, the cluster-head election makes use of a *stability index*. This index is computed from local hardware information (e.g., battery percentage, or the device's resources) which tries to indicate the probability of a node leaving the network voluntarily (e.g., through movement) or involuntarily (e.g., through shutdown due to low battery). This information is then piggybacked in Thyme's maintenance messages (i.e., periodic beacons), so that a node may compare its index to that of its neighbors.

At each cluster data dissemination cycle, each node compares its stability index against that of its neighbors: the node with the highest value starts the dissemination process. In the event that two or more nodes have the exact same stability index, we break the tie using their identifiers. A node that joins the network will wait for a predetermined amount of time to track the stability index of all other cluster nodes, before being able to join the pool of stable-ready nodes. Cluster data dissemination is confirmed by broadcasting the indexes' timestamp to all cluster members. The absence of this information, triggers a new cluster-head election and the resend of the cluster data.

Since the described process has weak and eventual coordination properties, it is possible for more than one node to establish itself as the cluster-head. To avoid duplicating and skewing cluster data information, we use back-offs to desynchronize the cluster data emission, and the edge server keeps track of the latest received timestamp for each cluster.

6.4.9 Dealing with Mobility and Churn

Mobility within a region is not disruptive, as long as the device remains connected to the AP, since the clustering process is not bound to location information. However, mobility may cause a client to leave a region. When so happens, the client's absence is eventually detected, when another mobile node or the server attempts to contact the absent client for data notification or retrieval purposes. In such case, the node's persistent publications will still be managed by the system, as long as there are resources to do so. However, subscriptions will eventually be discarded, and this information later propagated to the server as part of the cluster data. From that point onward, the node's subscriptions will no longer be processed in the abandoned region.

If the absence is temporary, it may not have been detected and everything continues as before. Otherwise, it is as if the node is entering a new region. In such situation, the node's subscriptions must be (re)installed in the entering region. This is done automatically as soon as a server is found. To avoid receiving duplicate notifications of previously published data, the lower end of the time interval is internally updated to the one of the latest publication received.

6.5 The Anatomy of GardenBed's Caches

From Thyme GardenBed's perspective, namespaces are distinct execution contexts. Thus, in order to provide their service to all the connected clients, independently of their namespace, servers adapt to the size and needs of each namespace's population.

For that purpose, we developed a namespace-sensitive caching model that we called *adaptive multipart caching (AMC)*. Contrary to a single cache, AMC is an array of caches, one for each namespace within the region, as depicted in Figure 6.7. The goal is to relax the management and size of the caching space assigned to each namespace. If the overall needs of a namespace changes (e.g., due to an increase of its population) and requires an increase in cache size, AMC will adaptively fulfill such request without ever exceeding the configured maximum global cache size, decreasing the size of the other caches, if needed. The repartitioning mechanism calculates the fraction among the namespaces' expected cache size and the whole expected cache size, if no limits were imposed to the number of entries being inserted in both cases, and then scales this value to the effective (and configurable) AMC total cache size.

One of the main advantages of this model is that it dampens the negative impact of scenarios where there is a short burst of a large amount of `retrieve` operations coming

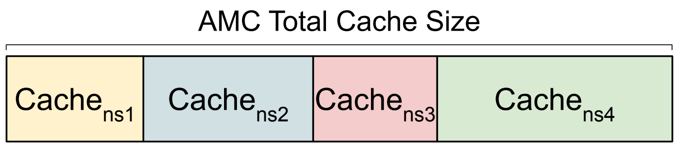


Figure 6.7: Adaptive multipart caching structure used in THYME GARDENBED.

from a single namespace. Although this could simply be a non-recurring and one-time event, the rising popularity of the target items frequently leads to a considerable amount of new cache entries, which could result in an overshadowing and ultimately a complete eviction of the previously cached data. The AMC approach smooths this process, preventing short popularity bursts to strip away a big amount of cache space in one sitting. If, indeed, this popularity trend continues, AMC will gradually increase the namespace's cache size, and with that fit the needs of the clients. For the specific cache implementation we used Caffeine [80].

The LPC is composed of a single AMC. Although the items to be cached are retrieved from the mobile devices one at a time, their insertion in the cache is periodic and performed in batches. This approach is a trade-off between the number of times the AMC repartitioning is executed (which is an expensive operation) and the time it takes for an item to be available from the edge. Figure 6.8 depicts how the batch-insertion timer integrates with the timer that triggers the ranking algorithm.

In turn, the PreC is the only one that does not make use of the AMC model. This is because *Provisioning the Subscription Catalog* (§6.4.4.2) is already namespace-sensitive, and thus the use of AMC would be redundant. Instead, the PreC comprises two one-dimensional disjoint caches: one for the metadata and another for the data. Having distinct caches with configurable sizes increases and encourages the *discoverability* of new items created throughout the entire system. The insight is that, since metadata are expected to be considerably smaller (when compared to data entries), the metadata cache can store a lot more entries before reaching the same memory footprint of the data cache. Consequently, if a data entry is evicted but the associated metadata entry is still cached, it is still possible for a client to obtain that data item: the server verifies the requested

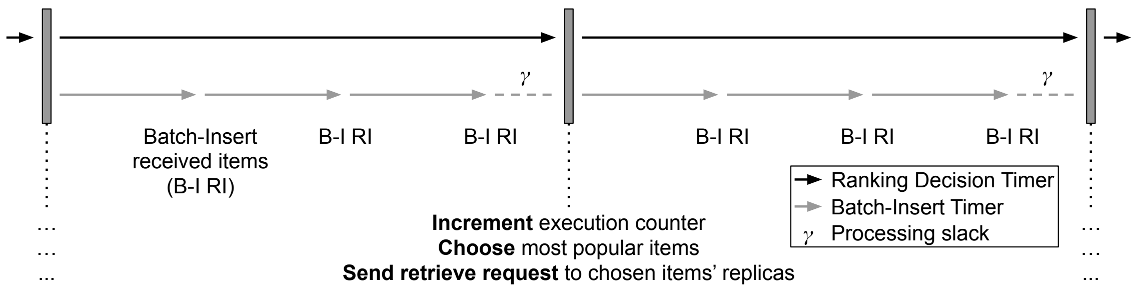

γ
γ
Batch-Insert received items (B-I RI)
B-I RI
B-I RI
B-I RI
B-I RI
B-I RI
Ranking Decision Timer
Batch-Insert Timer
γ Processing slack
Increment execution counter
Choose most popular items
Send retrieve request to chosen items' replicas

Figure 6.8: AMC timer scheme for inserting items in the cache used in THYME GARDENBED [278].

Table 6.1: Mobile devices specifications for Thyme GardenBed Android experiments.

	Motorola Nexus 6	Motorola Moto G (2nd gen.)
CPU	Quad-core 2.7 GHz	Quad-core 1.2 GHz
RAM	3 GB	1 GB
Storage	32 GB	8 GB
Battery	Li-Po 3220 mAh	Li-Ion 2070 mAh
Wi-Fi	802.11 a/b/g/n/ac	802.11 b/g/n

data is not stored locally, checks the associated metadata and, based on that, routes the retrieve request to the source region.

Lastly, the ORC's structure is similar to the one of the PreC, however each of the disjoint caches—one for metadata and another for data items—uses the AMC model. Unlike the LPC, items are inserted individually as they arrive, rather than in batches. Thus, the repartitioning workflow is only triggered after a configurable number of inserts have been performed in one of the caches.

6.6 Evaluation

To fully evaluate the behavior and workflows of Thyme GardenBed, we segmented the testing scenarios into two distinct environments: real world and simulation experiments.

6.6.1 Real World Experiments

Thyme GardenBed can be used to implement a plethora of use cases and applications that run at the edge (see §7). One specific use case is a photo gallery application (§4.6.4) that allows users to share photos without resorting to centralized services [47]. This *app* supports the subscription of tags, and the publishing and download of photos in different galleries. In this real world assessment, we utilized several Android smartphones described in Table 6.1: three Motorola Nexus 6, and three Motorola Moto G (2nd gen.), all with Android 7.1.1.

Moreover, in order to grasp how the system would perform if executed in a real world equipment, we used a computer laptop running an instance of the GardenBed server, directly connected to a consumer-grade wireless router. We also limited the number of cores used by each GardenBed server instance to 16 [1].

6.6.1.1 Latency versus the Cloud

One of the goals of Thyme GardenBed, as well as other edge-related systems, is to provide a better user experience to the nearby clients by having higher performance and lower latencies, when compared to systems deployed solely in the cloud. Therefore, since we are expecting retrieve operations to dominate the mobile clients' traffic, we compare our system's latency against a cloud infrastructure, when answering to retrieve requests

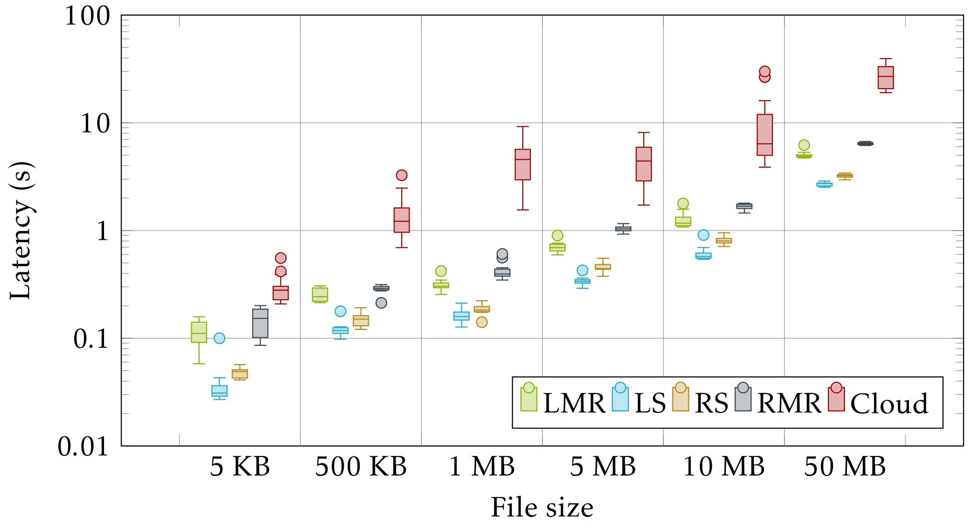


Figure 6.9: Retrieve operation latency in Thyme GardenBed [278].

from the mobile clients. To set up this evaluation scenario, we stored files of different sizes in the Amazon web services (AWS) [2] S3 cloud storage service (using the closest data center to our setup, mounted in our department building).

To evaluate the cloud counterpart, we downloaded each file multiples times, at various times of the day, and computed the average measured latency. Regarding our system, we measured the retrieve latency for all the possible scenarios:

Local mobile replica (LMR) - the retrieve request is sent to a mobile node that is currently replicating the item within the same region;

Local server (LS) - the operation is served directly by the local edge server;

Remote server (RS) - the item is not replicated by a neighboring mobile node nor by the local edge server, but is in the LPC of the server where the item was originally published, i.e., the retrieve request is routed from one edge server to another; and

Remote mobile replica (RMR) - the same as before, but the item is not in the server's LPC and hence must be obtained from a mobile node in another region.

The results across all scenarios are depicted in Figure 6.9. We may observe that, in Thyme GardenBed, retrieving a data item from the local or remote servers present the best results, and these are relatively close across all file sizes. This is expected since inter-server communication is usually much faster than Wi-Fi communication. We may also see that our approach provides lower latencies than the cloud infrastructure, when downloading the same file, and this gap widens as file size grows.

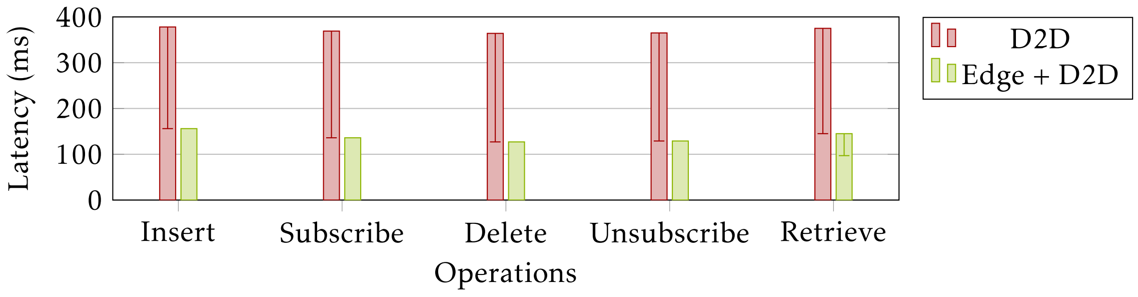


Figure 6.10: Operations latency in THYME GARDENBED [278].

6.6.1.2 Latency versus Only D2D

Given that latency is a critical metric in user-centric services, we delve further into this element and quantify the speedups awarded by the edge servers wrt. each THYME operation, when compared with the system using only D2D. Here, we use small data items—with 64 bytes for both the photo and its description—to determine the latency baseline without getting much affected by the transfer time. Furthermore, we only utilized the Motorola Nexus 6 smartphones to estimate the operations' latency in order to minimize disparities among hardware readings.

Figure 6.10 showcases the average measured latency of all operations. For D2D, the base values represent the latency when the sender is two hops away from the operation receiver, while the negative error bar shows the value for one-hop communication. In turn, the error bar in the Edge + D2D retrieve operation depicts the latency when the request is sent to the local server instead of a mobile node.

From this chart stems the fact that multi-hop routing has a considerable impact on the overall latency of an operation, resulting in a 170% increase, on average. Although the routing procedure is logically non-demanding, its workflow is almost entirely I/O bound, which contributes, on a higher degree, to the delay. When eradicated the routing middleman, D2D falls back to a mode similar to our approach, where peer-to-peer communication channels are established directly. In such context, the introduction of edge resources and additional edge logic produces latencies equivalent to the ones obtained in the D2D best case scenario (i.e., one-hop).

In the particular case of the retrieve operation, THYME GARDENBED brings considerable speedups when the request is served directly by the local server, leading to a ∼33% latency reduction. From all operations, insert and retrieve are the ones more susceptible to file size, as the file and its description are transferred, resulting in increased traffic, and therefore larger latency. The remainder present overall similar results, as they are independent from the file size.

We now further test the retrieve operation with different file sizes, in Figure 6.11. As estimated, the retrieval latencies for the 500 KB, 1 and 5 MB files follow approximately the 170% additional delay when comparing the two-hop routed workflow against the D2D approach. Similarly, we verify that the latency difference by getting the item from

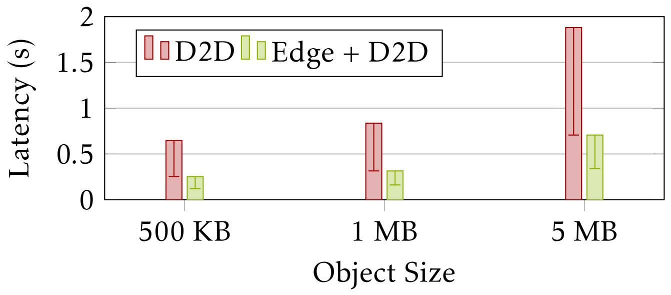


Figure 6.11: Retrieve latency varying object size in Thyme GardenBed [278].

the edge server is consistent with the initially recorded 33% decrease. Thus, to retrieve a 5 MB file, a client using Thyme GardenBed will take around 0.7 seconds (and approximately 0.3 seconds if the item is available from the edge), which we argue to be completely reasonable for real-time, interactive applications.

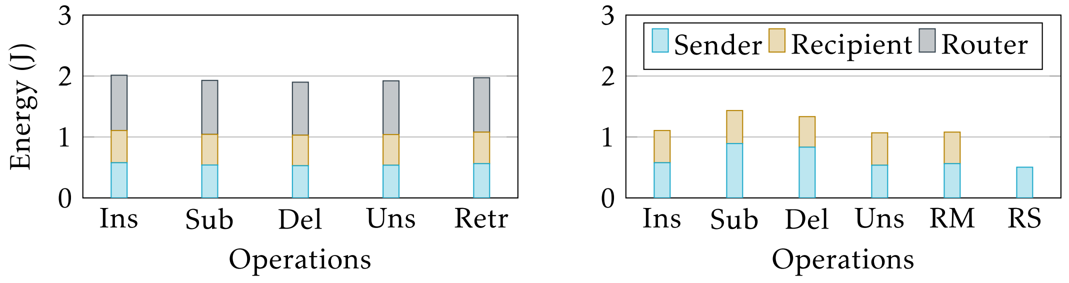
6.6.1.3 Battery Consumption

Device battery consumption is also a critical metric that needs to be taken into account, particularly in mobile edge systems that crowd-source resources from the mobile clients. To acquire a perception on the impact of using the edge infrastructure, we measured the battery footprint—in Joule—needed for a mobile device to execute each operation in both system configurations: Edge + D2D and only D2D. However, since the system is comprised by a set of nodes working cooperatively, we tracked not only the sender's battery overhead but also that of every node essential to the transmission and processing of an operation. As in the previous section, we utilized 64 byte photos and descriptions[2], and a single edge server. Moreover, these tests were limited to the Motorola Nexus 6 smartphones to reduce the hardware variance.

Figure 6.12a shows the average battery utilization using only D2D, from the standpoint of the sender, a single router, and the recipient that processes the request (and sends the response back to the sender). We verify that all operations carry a very similar cumulative battery footprint, close to 2 Joule (about 0.0045% of battery for the Motorola Nexus 6). Individually, the sender uses an average of 0.55 Joule while the recipient consumes 0.51 Joule. On the other hand, the router utilizes approximately 0.89 Joule, which makes up the bulk of the entire operation's cumulative battery consumption, at 46%. Also, the router battery usage component is foreseen to grow somewhat linearly with the amount of intermediate hops.

Next, Figure 6.12b presents the results for Edge + D2D (RS and RM are retrieve operations replied by the server or by a mobile node, respectively). Due to the absence of multi-hop routing, the cumulative metric only comprises the sender and recipient. Consequently, when multi-hop routing comes into play, the edge-backed solution presents a

[2]Naturally, as Figure 6.11 shows, larger image sizes mean higher latency and proportionally higher battery consumption.



(a) D2D. (b) Edge + D2D.

Figure 6.12: Operations' energy usage in Thyme GardenBed [278].

much lower battery utilization for every operation (with a 37% improvement on average). The lower-end is improved even further when a retrieve is successfully processed by the edge server, thus eliminating the recipient from the equation, and becoming the least resource-intensive operation, using a total of 0.5 Joule. On the other hand, the downside of including the edge resources is most noticeable on the subscribe and delete operations, as they were modified to also send the message to the server, leading to a battery consumption increase of approximately 61% on the sender.

The presented measurements 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 around 440 Joules) and showcase the results in Table 6.2 (the higher the values, the better). 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 Thyme GardenBed can be used throughout small to medium duration events without presenting an excessive energy consumption to the users' devices.

On the other side of the spectrum, Figure 6.13 illustrates the battery consumption of operations that are specific to Thyme GardenBed: finding the nearest server through

Table 6.2: Number of executed operations to reach a 1% battery consumption in Thyme GardenBed (*left - sent to a mobile node; right - sent to the edge server) [278].

		Insert	Delete	Subscribe	Unsubscribe	Retrieve
Sender	D2D	761	832	815	819	782
	Edge + D2D	761	528	493	819	782/873*
Router	D2D	481	506	497	499	493
	Edge + D2D	—	—	—	—	—
Recipient	D2D	833	878	875	875	851
	Edge + D2D	833	878	812	829	851/—*

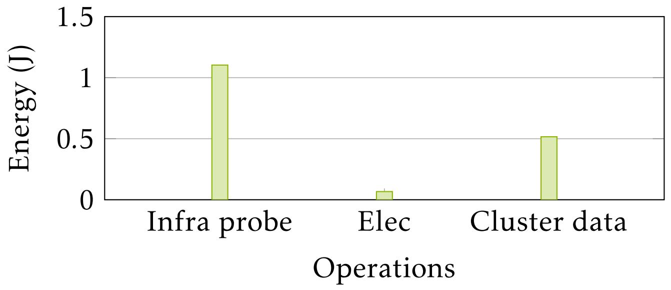


Figure 6.13: Energy usage for edge-specific operations in Thyme GardenBed [278].

multicast probing, cluster-head *election*, and emission of *cluster data* (only cluster-heads). From this plot, we can identify that the "Infrastructure Probe" workflow is the operation that requires the largest amount of battery. This is due to the Android-related mechanisms for handling multicast, which can cause a noticeable battery drain[3]. Yet, since this operation is only executed once when a mobile node joins the network, we consider its impact to be negligible.

Concerning cluster data dissemination, the vast majority of nodes within the system will simply execute the election process during each dissemination round, while only one per cell will go further and disseminate the actual data to the server (i.e., the cluster-head). To consume 1% of the device's battery: 1) a non-cluster-head node has to process approximately 6500 rounds; and 2) a cluster-head node has to disseminate cluster data around 853 times to a server. In the end, we argue that the overheads caused by these edge-only additions are completely acceptable due to the minimal battery consumption and the benefits they bring.

6.6.1.4 Overloaded Clusters

A disadvantage of Thyme's P/S (cluster-based) approach lies when only a few topics/tags are popular. For instance, during a football match in a stadium, users are expected to subscribe and publish most of the data on tags related to the name of the playing teams or tag "goal". Consequently, the majority of the operations will be disseminated to the cells that are indexing those topics, resulting in an overload of the nodes residing within those cells, while leaving others with a much lower workload. 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, we assess the benefits of using the edge infrastructure in these skewed cases. To achieve this, we implement a scenario that mimics a real-life event at a football stadium: after a team scores a goal, 100 new items with the topic "goal" are inserted, and users issue subscriptions for that topic, one in every second, during a total of 300 seconds.

The results in Figure 6.14 confirm one of the benefits of having edge servers. As a region's server continuously caches the generated (popular) items, the subscribers start to

[3]`https://developer.android.com/reference/android/net/wifi/WifiManager.MulticastLock`

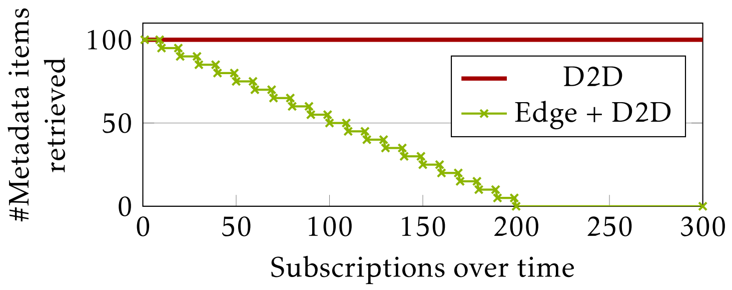


Figure 6.14: Notifications sent from mobile nodes in Thyme GardenBed [278].

receive less and less notifications through mobile means, as these are increasingly coming from the local server.

Figure 6.15 transposes the gradual mobile traffic reduction with the addition of the infrastructure resources, into battery utilization, showing a 44% reduction for Thyme GardenBed.

In conclusion, we are able to drastically reduce the mobile nodes battery consumption with 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's battery savings to grow linearly with the total amount of published items that are stored in the infrastructure. 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.

6.6.2 Simulating Mobile Devices

In order to assess the scalability of Thyme GardenBed without hardware limitations, we resort to the simulation of the mobile nodes. To that end, we developed a custom trace-based simulation framework that offers a total rework of the system's networking layer, to support logical dissemination of messages among any number of virtual nodes. We simply deployed the required number of edge server instances and connected all the virtual nodes to them, in order to define the regions.

Here, we use a university scenario: half of the students attend a course lecture, followed by an intermission and, after that, the rest of the students attend the course's

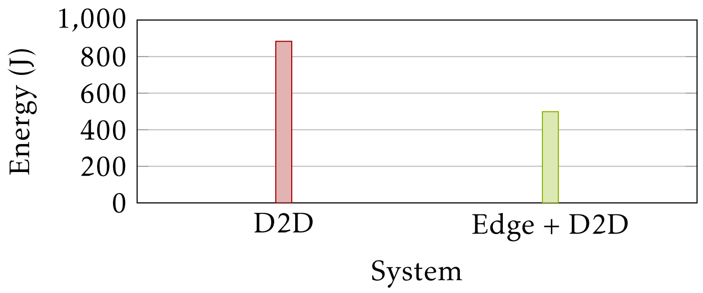


Figure 6.15: Total energy usage on over-utilized nodes in Thyme GardenBed [278].

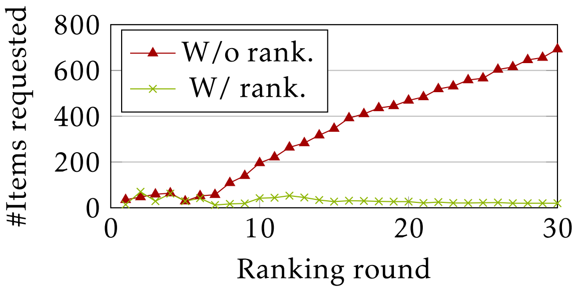


Figure 6.16: Number of data items requested by the server to mobile clients per popularity decision round in Thyme GardenBed [278].

second daily lecture of the same class. During each lecture, 15% of the attending students subscribe to all the lectured topics right at the beginning and are more predisposed to inserting items throughout the entire lecture (e.g., the "good" students). The rest of the students have a lower interaction rate and will only start to contribute after a predetermined amount of time into the lecture (e.g., the "regular" students). Throughout the entire duration of the trace, all users will be subscribing and inserting items with other topics that are relevant and global to the entire university. Each lecture is 45 minute long with a 10 minute intermission, totaling 100 minutes. For most of the students, subscriptions were issued randomly throughout the lecture until all of the topics were met. Insert operations were generated with a rate of 30 and 5 per user per hour for good and regular students, respectively. Regarding university related topics, subscriptions were generated with a rate of 10 per hour, for all students, while insert operations were delivered with a rate ranging from 10, for students outside the lecture, and 2 per hour, for students currently attending the lecture. To further simulate realistic interactions and reaction times, retrievals are issued with a 70% probability and within 30 seconds of the notification arrival. All tests were executed with 100 virtual nodes per region.

6.6.2.1 Ranking Algorithm

Here, we evaluate the advantages of filtering data requested by the server, i.e., popularity ranking. For that, we track the number of retrievals initiated by the server to populate its LPC, as well as the latter's hit ratio. To that end, we deployed a single GardenBed instance and analyze the first 30 ranking rounds.

Figure 6.16 presents the number of item requests made by the server during each ranking round, in both scenarios. While the first few rounds are similar (i.e., both are warming up the LPC), subsequent rounds for the scenario without the ranking algorithm show a constantly increasing amount of item requests that follows the rate of which new items are inserted. The use of a ranking mechanism stabilizes the number of requests after the LPC is full. Onward, only small adjustments to the LPC's contents are made (never requesting more than 70 items at any given round).

Moreover, Figure 6.17 shows that the LPC hit ratio when using the ranking algorithm

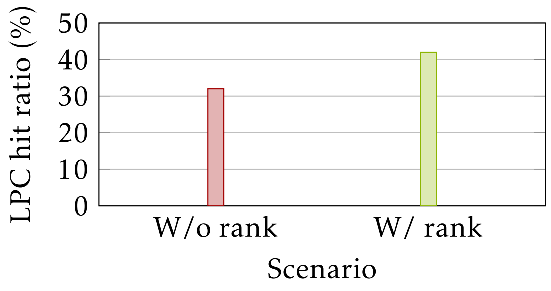


Figure 6.17: Popularity ranking and the LPC hit ratio in Thyme GardenBed [278].

presents far better results, namely a 10% increase. Thus, we conclude that even with our coarser approach for calculating popularity ranking, we are able to provide a better storage utilization through an acceptable amount of requests per ranking round. We can further correlate these findings with a lower battery and processing utilization from the mobile nodes, as we are able to reduce the amount of items that are mindlessly offloaded into the edge, while still successfully serving 44% of the received retrieve requests, which would have to be processed by mobile replicas otherwise.

6.6.2.2 Consumption Notification Optimization

Here, we evaluate the potential benefits of using the proposed *Consumption Notification Optimization* mechanism (§6.4.5).

Figure 6.18 shows the comparison between using the consumption notification approach versus no notification, in terms of the number of items that are sent from a server to another during each *data provisioning* round. As we demonstrate, the number of items sent between edge servers during every *Provisioning the Subscription Catalog* phase, after the first execution, is always lower when using the proposed mechanism. On average, this translates in a reduction of approximately 40% of items sent per round.

6.6.2.3 LPC Batch-Insert

Here, we analyze the behavior of the LPC batch-insert mechanism described in §6.5, in order to evaluate its trade-offs. We compare the batch-insert approach (described in

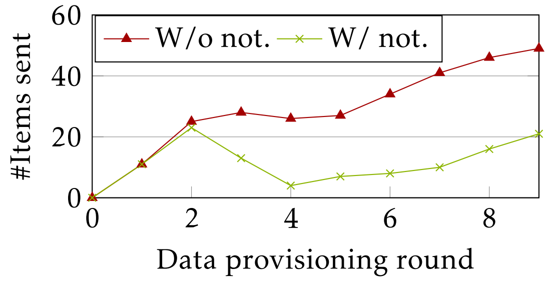


Figure 6.18: Impact of consumption notification in Thyme GardenBed [278].

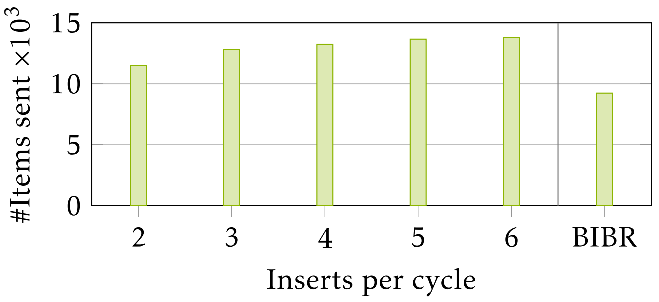


Figure 6.19: Number of items served by the edge server in Thyme GardenBed [278].

Figure 6.8) against another approach that batch-inserts all received items right before the next execution of the ranking algorithm (*BIBR* in the plots). Another alternative is to insert the items in the cache individually, as they arrive at the server. In fact, up until the moment the cache is full, this approach serves more retrievals from the edge than the alternatives, since items are available to the users sooner. However, with a full cache, inserting items one by one renders the AMC mechanism futile, and hence precludes the cache's dynamic space management.

The setup for this experiment used one server running in a Intel® Xeon® E5-2620 v2 @ 2.10 GHz and 64 GB of RAM. We kept track of the time the server spent processing the LPC's AMC, as well as the number of retrieve operations it served (triggered by the mobile nodes). The period of the ranking algorithm timer (i.e. the cache update period) was set to 30 seconds. The number of batch-inserts per ranking round ranged from 2 to 6. Bigger values would degenerate in the "insert on arrival" scenario.

Figure 6.19 exhibits the number of download requests served by the server, for both approaches. Being bound to a 30 second cache update period, BIBR performs naturally worse, with the server only serving ~9 000 items to the clients. The batch-insert approach, even with only two inserts, increases the number of items sent in ~2 500, which is roughly a 25% improvement. Onward, performance increases at a slower rate, going from ~11 500 items in the 2-batch approach to ~14 000 with 6 batches.

On the other hand, in Figure 6.20, we observe the CPU time needed to process the AMC algorithm. Here, the increased visibility provided by our solution correlates with a higher processing overhead, that grows sub-linearly with the amount of batches. In the BIBR approach, the AMC processing overhead was ~500 ms, while when batch-inserting 2 and 6 times per round, this accounted for 515 and 729 ms, respectively.

In the end, this presents a trade-off that the application developer will have to analyze in order to fine tune the system for its needs and achieve a better performance. For instance, if a considerable amount of items are published every ranking round, leading to a higher content turnover in the caches, the developer will be able to adjust the properties for this workflow and increase the number of batches per round in order to consequently increase the visibility of newly created items.

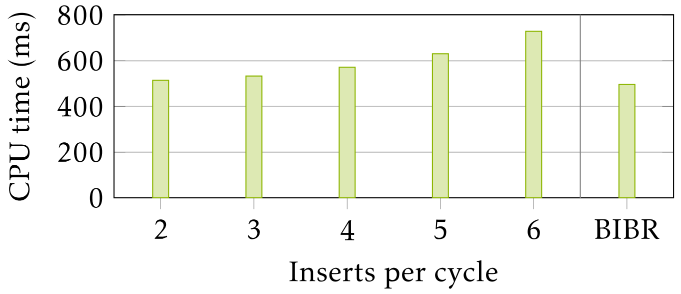


Figure 6.20: CPU time spent on the AMC used in Thyme GardenBed [278].

6.7 Concluding Remarks

In this chapter, we present Thyme GardenBed, a framework for content storage and sharing for mobile devices in multi-region edge networks. It cooperatively and symbiotically leverages both D2D interaction and edge servers to allow the flow of content in networks of mobile devices spanning across edge network regions. We further detail Thyme GardenBed's mechanisms and its intricate caching model. At a high level, this work can be seen as a data storage and dissemination system for a wide-area setting, like a campus, a music festival, or an amusement park.

To fully evaluate the behavior and workflows of Thyme GardenBed, we segmented the testing scenarios into two distinct environments: real world and simulation experiments. Overall, our evaluation shows low response times, allowing interactive usage, and low energy consumption, thus supporting its usage in a variety of events without excessive battery drainage. Furthermore, compared to a cloud solution, Thyme GardenBed yields considerable latency speedups.

6.7.1 Discussion

In general, Thyme GardenBed makes a fundamental *energy shift*. With GardenBed, we introduced edge resources into the system, which can be inherently more powerful and capable than client devices (mainly regarding energy). This way, it allows us to optimize and *offload* a portion of the requests' processing and data from the clients into the stationary devices. Ultimately, saving energy in the mobile devices and lowering their processing overhead. In the end, some of the energy that before was expended by the mobile devices for processing requests and managing the system is now shared between the devices and the edge servers, creating a symbiotic relation among them. Since edge servers usually can have unlimited energy, this overhead shift can be very desirable, i.e., the added system complexity is offset by the shift in energy expenditure (from the mobile devices to the edge servers).

Furthermore, because edge servers can be connected through wired means to other servers, we essentially overcome the mobile devices' wireless range limitations, enabling the possibility of distant clusters of users to interact with one another indirectly. On the other hand, by having users take an active role in the system, we are able to minimize the chances of the infrastructure being overloaded or a single point of failure.

6.7.2 Future Work

Thyme GardenBed is already a fully-fledged system, however there is always room for improvements. As future work, we highlight the following directions: the quantitative comparison with other edge-caching works from the literature; an adaptive (and lightweight) ranking algorithm with a decaying function; allow a dynamic membership of edge servers; and inter-region node mobility-awareness, leveraging devices' mobility to opportunistically propagate content to other regions.

6.7.3 Publications

The work reported in this chapter resulted in the following publications:

- **Data Storage and Sharing for Mobile Devices in Multi-region Edge Networks** [246]. *João A. Silva*, Pedro Vieira, and Hervé Paulino. In Proceedings of the 21st IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (**WoWMoM**). Online, 2020.

Chapter 7

Beyond Thyme: The Edge Garden Ecosystem

"The true sign of intelligence is not knowledge but imagination."
— Albert Einstein

During the thesis, the developed work evolved and forked into multiple research directions pursued by different Master students (and myself), in what we called the *Edge Garden* ecosystem. The majority of these works are based on and derived from Thyme (§4), and also Thyme GardenBed (§6). Some of them are Master's theses (some concluded and others ongoing) in which I have been (and are) involved, i.e., in which I collaborated and supervised.

In the first two sections, §7.1 and §7.2, we present Ephesus and Jumper, respectively, works that I pursued before Thyme. Then, in the next five sections, §7.3, §7.4, §7.5, §7.6, and §7.7, we present Oregano, Basil, PS-CRDTs, Peppermint, and Wasabi, respectively. These are all concluded M.Sc. theses. In the remainder sections, §7.8, and §7.9, we describe Chives, and Basilicum, respectively, both M.Sc. theses that are still ongoing. We conclude the chapter with §7.10, by presenting our final considerations regarding the featured works.

7.1 Ephesus: Ephemeral Storage for Mobile Devices

This work resulted from the Master's thesis of Ricardo Monteiro [177], entitled "Distributed Storage in a Cloud of Mobile Devices", which was later extended and reworked by myself.

As already mentioned, mobile devices are a big source of user-generated content, and users want to share that content among them. Thus, Ephesus appears as an alternative to typical data sharing solutions among mobile devices, such as centralized systems (e.g., Dropbox) or pair-wise device-to-device (D2D) file exchange applications. It is an ephemeral distributed data storage system for networks of mobile devices, where users can asynchronously publish their own files and obtain files that have been shared by others. The *ephemeral* attribute emphasizes the fact that an instance of the system will only exist as long as there are devices supporting it. It is entirely supported by a set of interconnected mobile devices, thus not requiring any kind of Internet access. Hence, as soon as the last device disconnects, the system will cease to exist and the data it stored will no longer be accessible.

The system provides a key-value store interface with operations `put`, `get` and `remove`, and is extended with a `list` operation. The latter inquiries the storage's global state, which in the mobile wireless environment may incur in considerable latency and energy consumption. To reduce both latency and energy consumption, it employs a lightweight best effort and eventually consistent listing mechanism. In turn, to meet the volatility of the target environment, the solution is designed to be resilient to churn, i.e., the ingress/egress of devices into/from the system, to some extent, and also to leverage data replication in order to guarantee data persistence and availability in a best effort way. Additionally, energy limitations of the individual devices are taken into consideration and the system is designed to be energy-aware and fair in the amount of consumed resources from each participating device.

A prototype of the solution was developed in Java, based on a preexisting distributed hash table (DHT)[1], and used to build a photo sharing Android application, as a case study. The application runs in out-of-the-box Android devices without the need to root the operating system. Overall, the experimental results show that Ephesus may be used continuously during an event, such as a party, a concert or a sporting event, without exhausting the device's battery. It effectively allows users to share contents in an easy way, and also provides reasonable response times perceived from the users' perspective.

In summary, the contributions of this work are as follows: i) the proposal of an ephemeral distributed data storage system for mobile devices in close geographical proximity that does not require Internet access; ii) the proposal of a lightweight mechanism for listing the data files available in the system; iii) the experience report of an Android prototype with a case study that makes use of the proposed system; and iv) the characterization of the scenarios better suited for the use of the proposed solution.

This work resulted in the following publications:

- **Ephemeral Data Storage for Networks of Hand-held Devices [243].** *João A. Silva*, Ricardo Monteiro, Hervé Paulino, João M. Lourenço. In Proceedings of the 14th IEEE International Symposium on Parallel and Distributed Processing with Applications (**ISPA**). Tianjin, China, 2016.

- **Armazenamento Distribuído para Redes de Dispositivos Móveis [179].** Ricardo

[1] https://tomp2p.net/

Monteiro, *João A. Silva*, João M. Lourenço, Hervé Paulino. In Proceedings of the 7th Simpósio Nacional de Informática (**INForum**). Covilhã, Portugal, 2015.

- **Decentralized Storage for Networks of Hand-held Devices [178].** Ricardo Monteiro, *João A. Silva*, João M. Lourenço, Hervé Paulino. In Proceedings of the 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (**MobiQuitous**). *Poster paper*. Coimbra, Portugal, 2015.

7.2 Jumper: Opportunistic Combination of MANETs and Infrastructure

This work was done by me, in parallel with the development of Thyme.

Mobile ad-hoc networks (MANETs) are formed dynamically by mobile nodes that are connected wirelessly without resorting to a pre-existing network infrastructure. Thus, interaction among nodes is achieved through the wireless broadcast medium without any central coordination entity (like in a peer-to-peer (P2P) fashion). Nodes can move freely, thus the network topology may change rapidly and unpredictably. Furthermore, the lack of a central coordination entity makes routing messages in MANETs a challenging task. Nodes lack a global, consistent, and up-to-date knowledge of the network topology, being required to make routing decisions based only on local (and potentially partially incorrect) knowledge.

However, given the increasingly ubiquitous Internet access through other networking technologies that co-exist alongside ad-hoc networks (e.g., Wi-Fi and 3G/4G networks), some of these nodes might also have simultaneous access to a network supported by infrastructure. This uncovers several opportunities when devising routing strategies, allowing the opportunistic combination of ad-hoc networking with infrastructure access. Therefore, when routing messages, two approaches can be employed: one entirely in the ad-hoc network, and a second one that also makes use of the access to the infrastructure. So, although the ad-hoc network must be entirely self-supporting (e.g., for emergency or disaster situations), it can leverage the infrastructure (when present and even if intermittently) during normal operation.

Mobile devices, such as smartphones or tablets, are natural examples of nodes that may have simultaneous access to both the Internet and to an ad-hoc network. In the last years, the proliferation of this kind of devices, along with the increasing growth of their capabilities, has spawned research on the adaptation of MANET techniques for the mobile devices world (e.g., [70]). In this exploratory work, we argue for the opportunistic combination of ad-hoc networking with infrastructure access, exploring a way of capitalizing on this double access and enabling possible optimizations, such as improving communication and energy efficiency in mobile networking.

During the process of forwarding messages, the routing protocol has to decide which alternative to use: only through the ad-hoc network; or using the infrastructure access, making messages "jump" through the (ad-hoc) network. Thus, we propose Jumper, a decision algorithm that determines the best path for each message, entailing a possible decrease in latency, by avoiding the long ad-hoc hop-by-hop routing. Even otherwise, benefits may arise from the reduction of the overall aggregate energy costs of routing the message through all the intermediate nodes in the ad-hoc network multi-hop path. In the end, Jumper addresses the interaction between the ad-hoc routing and the infrastructure access by determining: i) when it is better for a message to be routed through the network using ad-hoc techniques; and ii) when it is better to route the message through a tunnel where the endpoints are nodes with access to the infrastructure, enabling long "jumps" over the network.

In summary, the contributions of this work are as follows: i) the proposal of a routing scheme opportunistically combining ad-hoc routing with infrastructure access; and ii) the algorithmic framework of Jumper, a decision algorithm determining the best path for each message, taking into account several parameters.

This work resulted in the following publications:

- **Towards the Opportunistic Combination of Mobile Ad-hoc Networks with Infrastructure Access** [242]**.** *João A. Silva*, João Leitão, Nuno Preguiça, João M. Lourenço, Hervé Paulino. In Proceedings of the 1st Workshop on Middleware for Edge Clouds & Cloudlets (**MECC@Middleware**). Trento, Italy, 2016.

7.3 Oregano: Distributed Computing on Mobile Devices

This work resulted from the Master's thesis of Pedro Sanches [223], entitled "Data-Centric Distributed Computing on Networks of Mobile Devices", and was built on top of Thyme.

Together, the exponential growth in the amount and capabilities of mobile devices, the increase in the amount of user-generated data, and the ubiquity associated with these devices, makes it interesting to start thinking in a different use for them. Accordingly, they can begin to act as an important part in the computations required by more resource-demanding applications, rather than relying exclusively on external services (such as in infrastructure clouds). Considering all the above, it is pertinent to use the resources available at the network edge, i.e., in the mobile devices, and (partially or fully) process data closer to where it is generated and consumed. By processing data near its source, applications can be more responsive, while relieving some of the load from both cloud and network infrastructures.

This work proposes Oregano, a framework for data-centric distributed computing for networks formed exclusively by co-located mobile devices, without needing cloud services, and even being able to work without access to network infrastructures. Differently from current state-of-the-art, where both computations and data are offloaded to (worker) mobile devices, Oregano moves computations to where data resides, reducing significantly the amount of data exchanged. It is capable of processing batches and streams of data generated, and stored, by a cloud composed of mobile devices. Oregano presents a programming and execution model based on the manipulation of sets of data called mobile dynamic data sets (MDDs) [214]. MDDs are logical entities that comprise data items of a given type, characterized by a Thyme tag. They are stored in Thyme, and processed by a data-centric batch/stream computing model.

A possible use case for this proposal is a birthday party, where participants take photos and share them, so that others may collect the ones they have interest in. Initially all photos may be shared with a single tag, #PartyEmily, hence defining one MDD. As the party evolves, participants may ask for the photos of a specific person, by supplying a photo of the person as a query. The resulting set of photos will define a new MDD that may be retrievable and made accessible to all through a new tag, e.g., #EmilyResult.

Based on the experimental results, both on a real and simulated environment, Oregano is proven to support scalability, by benefiting significantly from the usage of several devices to handle computations, and by supporting multiple devices submitting computation requests while not having a significant increase in the latency of a request. It is also able to deal with reasonable amounts of churn without being highly penalized.

In summary, the contributions of this work are as follows: i) Oregano, a framework for data-centric distributed computing on networks of mobile devices, capable of processing batches and streams of data generated by the devices, without requiring access to network or cloud infrastructures; and ii) the evaluation of our prototype (for Android) in both real world and simulation scenarios.

This work resulted in the following publications:

- **Data-Centric Distributed Computing on Networks of Mobile Devices [224].** Pedro Sanches, *João A. Silva*, António Teófilo, Hervé Paulino. In Proceedings of the 26th International European Conference on Parallel and Distributed Computing (**Euro-Par**). Online, 2020.

- **Computação Distribuída em Redes Formadas por Dispositivos Móveis [225].** Pedro Sanches, António Teófilo, Filipe Cerqueira, *João A. Silva*, Hervé Paulino. In Proceedings of the 9th Simpósio Nacional de Informática (**INForum**). Aveiro, Portugal, 2017.

7.4 Basil: A Key-Value Store on Mobile Devices

This work resulted from the Master's thesis of José Afonso [4], entitled "Key-Value Storage for Handling Data in Mobile Devices", and was built on top of Thyme GardenBed.

Mobile devices are ubiquitous and are used in all sorts of different activities, constantly generating information that needs to be stored or processed somewhere. To cope with the huge amounts of data generated by these devices, traditionally applications resort to cloud services to provide them with the much needed computational and storage resources. However, these remote infrastructures still represent a large communication and processing hub. In turn, with edge computing, instead of processing and storing all data in distant cloud services, data (and processing) is spread among mobile devices and edge servers scattered in the network.

In order to fully take advantage of the low latency experienced in the edge, applications still need an *edge-oriented* distributed storage solution, capable of handling the contents generated by all of these mobile devices. The current state-of-the-art storage systems are able to provide these applications with a storing platform that uses mobile devices or edge servers as data storing points, but neither uses both.

This work proposes Basil, a key-value edge storage system, that uses both mobile devices and edge infrastructures as nodes of the system, capable of providing users from different locations with a cohesive and consistent distributed storage system. It basically provides a key-value store interface on top of Thyme GardenBed. As such, data resides on the actual devices, and its persistence is guaranteed by Thyme through passive and active replication mechanisms. Basil takes advantage of Thyme's tags to provide the mapping between keys and their respective values.

Differently from common key-value store, Basil provides a `list` operation, inquiring the system for a list of all stored keys. For this, it deploys an index in a specially reserved tag, where all used keys are inserted. Then, through Thyme's time-aware reactive storage (TARS) abstraction, nodes are notified as new keys are inserted in the system. As another feature, Basil also adds hierarchical keys, providing a namespace similar to a file system. It also provides a link and unlink operations (similar to the ones in file systems), where a new key is associated/removed with/from an already existing data item.

In the evaluation, Basil was compared with Cassandra [145], using the implemented use case—Class Quiz. Naturally, Basil reaps the benefits of its substrate, Thyme GardenBed, in terms of overall performance and data persistence. But as expected, the more powerful Cassandra infrastructure is able to outperform a cluster of weaker mobile devices for a single quiz instance. Nevertheless, as we increase the number of quiz instances, Basil's performance remains constant, while Cassandra increases its overheads in proportion to the number of instances. In the end, Basil can offer a flexible, stable and horizontally scalable framework for edge computing environments, independent from remote and/or local infrastructures.

In summary, the contributions of this work are as follows: i) Basil, a key-value edge storage system on top of Thyme GardenBed; ii) the design and implementation of a use case application for Android allowing students to participate in live class quizzes created by (authenticated) professors; and iii) the evaluation of our prototype in simulation scenarios, and its comparison against a fully-fledge key-value store.

7.5 P/S-CRDTs: CRDTs for Dynamic Environments

This work resulted from the Master's thesis of António Barreto [26], entitled "Conflict-Free Replicated Data Types in Dynamic Environments", and was built on top of Thyme allowing mutable data.

The implementation of collaborative applications in highly volatile environments, such as the ones composed by mobile devices, require low coordination mechanisms. The replication without coordination semantics of conflict-free replicated data types (CRDTs) makes these a natural solution, guaranteeing eventual consistency of the shared data. However, a limitation found on current CRDT models is the need for the knowledge of all the replicas whom the state changes must be disseminated to. This constitutes a problem since it is inconceivable to maintain said knowledge in a volatile environment where clients may leave and join at any given time and consequently get disconnected due to mobile network communication unreliability.

To allow for CRDTs to be effectively used in these environments, this work proposes the P/S-CRDT model that combines CRDTs with the publish/subscribe (P/S) interaction model, enabling the spatial and temporal decoupling of update propagation. Basically, the P/S system is used as the medium for propagating and sharing updates, and defines the update propagation pattern for shared CRDT objects. With this, CRDT update dissemination is completely decoupled from update reception, enabling its use in highly volatile scenarios where there is no total knowledge of all replicas. In a nutshell, updates are published to a CRDT object, stored by the P/S broker, and subsequently all subscribers are notified of new updates, retrieving updates and merging them with their replica of the CRDT object. We also present our versions of several CRDT synchronization models, namely state-based, operation-based, and Δ-based, describing the necessary adaptations and the specific requirements for each model.

We implemented the three proposed P/S-CRDT synchronization models on top of Thyme, and developed several CRDTs (e.g., counters, sets, maps). From the experimental evaluation, results show that P/S-CRDTs perform better than other CRDT models in volatile environments, requiring less bandwidth.

In summary, the contributions of this work are as follows: i) P/S-CRDTs, the proposal of an extension to the CRDT concept for dynamic environments, through the combination of CRDTs with the P/S interaction paradigm; ii) the specification of three P/S-CRDTs synchronization models; iii) the implementation of P/S-CRDTs on top of Thyme; and iv) the evaluation of the implemented prototype in simulation scenarios.

This work resulted in the following publications:

- **CRDTs em Ambientes Dinâmicos [27].** António Barreto, *João A. Silva*, Hervé Paulino, Nuno Preguiça. In Proceedings of the 11th Simpósio Nacional de Informática (**INForum**). *Short paper*. Guimarães, Portugal, 2019.

7.6 Peppermint: A Framework for Local Multiplayer Games

This work resulted from the Master's thesis of Salúquia Marreiros [167], entitled "A Framework for Turn-Based Local Multiplayer Games", and was built on top of Basil (also using features of PS-CRDTs).

Nowadays, mobile devices are a big part of people's lives, and are used from communication to a means of entertainment. In fact, mobile games (specially multiplayer games) are consistently on the top of app stores leaderboards, showing a clear growing trend.

Typically, multiplayer mobile games are played using centralized servers to store data and coordinate devices, which can be costly due to latency and battery usage, degrading the playing experience and diminishing engagement. This can be the most appropriate solution when players are distant from each other, but it is unnecessary when players are co-located. When in a purely local setting, as a way to diminish traffic to the Internet and have a better playing experience, we can take advantage of resources close to where games are being played. This rationale meets a recent trend of placing data and its processing closer to the users, in what is called the network edge.

This work proposes Peppermint, an event-based framework for turn-based local multiplayer games at the network edge, without requiring any centralized service. It allows programmers to easily develop turn-based games for multiple players in close physical proximity. Peppermint provides abstractions for *matches* and *players*. Thus, game developers only have to implement (and extend) certain parts of the framework with their game's logic, handling certain types of events triggered by the system. It supports the coordination of a match and its players (which may enter or leave at any time), the match state (e.g., scores and the board's current characteristics), modifications to the match's board (which may happen during a turn and need to be disseminated to the other players), turn movements, players' matchmaking, among others. Additionally, it allows for the co-existence of several matches (of possibly different games) concurrently in the system, and facilitates the retrieval of the set of existing matches when a player wishes to join one. To achieve all this, Peppermint leverages on Basil to associate players with matches, disseminate player's turn information to others, decide how to proceed when players enter and leave matches, and manage the game state accurately.

In summary, the contributions of this work are as follows: i) Peppermint, an event-based framework for the easy implementation of turn-based multiplayer games at the network edge; ii) the implementation of a case study, a distributed version of the Snake game; and iii) the evaluation through simulation of the implemented prototype.

7.7 Wasabi: Adaptive Replica Selection in Mobile Edge Networks

This work resulted from the Master's thesis of João Dias [74], entitled "Adaptive Replica Selection in Mobile Edge Networks", and was integrated into Thyme GardenBed.

With the ongoing increase of mobile devices, and application's growing reliance on the cloud, these infrastructures have become centralized hubs of computational processing and storage. With so much traffic being generated to and from these centralized infrastructures, network congestion and delays start to become more evident, specially for some types of applications. In turn, mobile edge computing (MEC) is a paradigm that aims to solve these limitations by bringing cloud services closer to mobile clients, effectively reducing end-to-end delays and saving backbone bandwidth.

Currently, many applications and services use replication to enhance their quality of service. Because content generated by mobile devices has a localized interest at first, data starts by getting replicated between these devices, and only when it starts to get popular is it eventually replicated (or cached) in edge servers. However, a problem arises when there is no replica selection mechanism for data retrieval. The resulting herd behavior can cause the network load to be poorly distributed, which combined with the unreliable wireless communication channels cause these systems to under-perform.

This work proposes Wasabi, a flexible replica ranking middleware for mobile edge networks. It works as a middleware service for client-server communication in edge network environments, that empowers clients with the necessary metrics to independently decide which should be the best replica to contact. Albeit it refers to clients and servers, these can be any node of the system, and a node can have both roles at the same time.

The middleware attaches to a data storage system, collecting and disseminating (configurable) network and computational-load metrics piggybacked in the system messages. Then, when a client requests a data item, Wasabi uses the gathered metrics, applies a ranking algorithm, and returns the best possible replica to request the data item from. It forms a continuous feedback loop between clients and servers in order to grant the former with a fresh (albeit usually partial) view of the system, using server-emitted, as well as client-observed metrics.

The middleware consists of the following high-level components: 1) a server-side aggregator component that collects system metrics on demand; 2) a metrics collector component configured to collect a metric value on demand and hooked into the aggregator; 3) a client-side replica classifier component that consumes metrics and is able to sort a set of replica nodes from most to least reliable according to the configured scoring logic; and 4) a client-side metrics observer component that can reactively compute metric values from system events.

We also propose MECERRA, a replica ranking algorithm specifically tailored for MEC, that addresses the challenges raised in these environments, such as churn, dynamic replicas, energy constraints, and metric freshness. It uses predefined network, resource-usage, and device-specific metrics to predict which should be the best replica to contact. In the end, Wasabi aims to decrease latency and boost both throughput and energy efficiency in the system, by avoiding over-saturated replicas. In this specific implementation, the middleware was integrated into Thyme GardenBed. Experimental results through simulation show that MECERRA finds the best replica much often than the alternatives, and Wasabi provides low overhead.

In summary, the contributions of this work are as follows: i) Wasabi, a flexible and system-agnostic replica ranking middleware for mobile edge networks; ii) the integration of this middleware within Thyme GardenBed; iii) MECERRA, our replica ranking algorithm tailored for MEC environments; and iv) the evaluation through simulation of the implemented prototype and the comparison of different replica ranking algorithms.

7.8 Chives: Dynamic Content-Based Indexing at the Edge

This work results from the still going Master's thesis of Cláudio Pereira, entitled "Dynamic Content-Based Indexing in Mobile Edge Networks", and is being built on top of the Oregano framework.

In the last years, there has been a huge increase in the usage of mobile devices, and also in the amount of user-generated content, e.g., photos, video, or messages. Usually, this data requires a permanent storage and its respective indexing in order for users to efficiently access it. However, due to the unpredictability of this data, a concern regarding its indexing starts to raise, as it can be hard to predict labels and indexes capable of representing every possible set of data.

For instance, during a birthday party, users may want to share photos and videos, which can be seen as uploading streams of data to a content sharing system. This stream will probably not have an index capable of representing its data, making difficult its retrieval as there is no semantic representation of such. However, as time passes, and the stream data continues to grow, we may be capable of predicting descriptive labels, thus allowing the indexing of this event. In order to successfully implement this strategy it would be required a framework capable of dynamically generating indexes, which is currently not available.

This work proposes Chives, a content-based indexing system for mobile edge networks, built on top of Oregano. It uses unsupervised learning clustering techniques to offer a novel data retrieval framework. The system receives data items and, using unsupervised learning, groups them (into clusters) based on a similarity metric. After a data set (i.e., a cluster) reaches certain requirements, it generates a human-readable label representing the data and indexes it for future retrievals, allowing the dynamic indexing of the uploaded content. For labels, the system evaluates the cluster's content and, using a popularity heuristic, generates a label capable of representing its content. Chives also offers two distinct methods for content retrieval: 1) content-based retrieval, which receives an image and in return provides similar images to the user; and 2) semantic-based retrieval, which receives a label and in return provides the indexed content mapping to that label to the user.

The system evaluation will use image data sets with human faces, and measure the system's response time, labeling accuracy, and other relevant metrics.

This is ongoing work, however, its current contributions are as follows: i) Chives, a content-based indexing system, built on top of Oregano, which uses unsupervised learning clustering to provide a novel data retrieval framework; and ii) the ongoing evaluation of the implemented prototype in simulation scenarios.

7.9 Basilicum: Basil in the Edge-Cloud Continuum

This work will result from the just started (and ongoing) Master's thesis of Francisco Nunes, entitled "Reactive and Persistent Storage for Mobile Edge Computing Environments", and is being integrated into Basil (and Thyme GardenBed).

The edge computing paradigm gave rise to a three-layered hierarchy, as depicted in Figure 1.1. At its base, we have the end-user devices which have somewhat limited resources but generate increasing amounts of data. In the second layer, we have the edge, a set of geo-distributed fixed servers located close to users, which allow data processing and storage with low latencies. Lastly, in the top layer, we have the cloud with its vast and powerful resources. Although systems do not always make use of the entire hierarchy, the advantages of an architecture that leverages all layers can be notorious.

In this work, we focus on use cases where users intend to use their mobile devices for sharing and viewing content generated in the context of their location. Thus, this work's main goal is to devise a system embodying the entire architecture depicted in Figure 1.2, integrating the three layers of the network hierarchy. We envision it to allow time-bound content with local interest to be shared and managed locally, using mobile devices and edge nodes, while data with global interest is persistently stored and managed in the cloud. This way, users can take advantage of the low latency in the local dissemination of content present in the devices, as well as resort to the edge nodes to establish contact with the persistent and global resources in the cloud.

Currently, Basil (§7.4) guarantees the storage and sharing of data in a local context using mobile devices, and uses edge nodes for caching popular content among regions. This work proposes to add a new level to this hierarchy, complementing it with a cloud infrastructure capable of offering persistence to data with global relevance. The goal is to bring the already existing P/S key-value data and interaction model to the cloud. This way, storage is not restricted to the limited resources of mobile devices, and their constant movement and possible exit from the system.

The lower layer—end-user devices—remains almost unchanged. Mobile devices are responsible for storing and disseminating data of local interest within a region. However, it will be necessary to enable users to define where data should be kept—either locally in the mobile devices, or globally in the cloud (or even in both). The same happens when the user queries data or subscribes to content under a key. It will have to make explicit where the operation should take effect. In the middle layer, edge nodes will maintain the previously developed caching service, but now they are also responsible for establishing the bridge between mobile devices and the cloud. At the top, the cloud, in addition to acting as a persistent data repository, it will also have to manage some information related to the P/S component (e.g., users' subscriptions).

This is ongoing work, however, its expected contributions are as follows: i) Basilicum, a data storage and dissemination system for mobile devices, leveraging in the entire edge-cloud continuum; ii) the development of the cloud component, and the integration of the three levels of the hierarchy into Basil; and iii) the evaluation of the implemented prototype in simulation and real-world scenarios.

7.10 Concluding Remarks

As written in the acknowledgements, no man is an island. In this chapter, we present several works exploring different research directions, that evolved from the solutions described in the thesis. These were works, some pursued by me (in collaboration with other researchers from the department), others pursued by different Master students in their theses. In all of them, I had the pleasure to collaborate and help in the students' supervision. These research works work in both ways. On the one hand, students help in pursuing several research ideas (with implementations, experimental evaluations, and reports) by executing their theses, also enriching this thesis in the way. On the other hand, these collaborations allow the development of other soft skills important in life (such as people management or task prioritization).

In the end, the works presented here show the flexibility and versatility of the TARS interface and of the works implementing it (i.e., Thyme and Thyme GardenBed).

Chapter 8

Conclusion

"True merit is like a river, the deeper it is, the less noise it makes."
— Edward Wood

This is the end, beautiful friend. This is the end, my only friend. This chapter closes the thesis. In §8.1, we summarize the main results presented in the thesis, and provide an answer to the questions introduced in §1. Lastly, §8.2 concludes the thesis by discussing some future research directions.

8.1 Conclusions

The thesis has proposed, developed, and evaluated solutions for data storage and dissemination in pervasive edge computing environments, either with or without access to network infrastructures. We illustrated the benefits of the proposed solutions through different types of evaluation, ranging from an analytical study to experimental evaluations using different kinds of simulation and also real-world devices.

The considered approaches can be divided into two parts: i) surviving without infrastructure; and ii) thriving with infrastructure.

First, since we address resource-constrained and highly dynamic heterogeneous environments, we favor a loosely coupled approach for data dissemination. Thus, we fuse the storage substrate with the publish/subscribe (P/S) paradigm and propose time-aware reactive storage (TARS), a reactive data storage and dissemination model. It provides persistent publications and allows queries (i.e., subscriptions) within a specific time scope. The insert operation of the storage substrate is merged with the publish operation of the P/S system, enabling applications to be notified as relevant data is generated and stored. Additionally, queries (i.e., subscriptions) are in the form of propositional logic formulas, and have a time frame defining when they are active. The innovative characteristics of TARS offer a novel way for sharing and accessing data that has been previously stored, or is being generated in quasi-real-time. In the end, this concept makes a fundamental overhead shift. It offers a reactive interaction model, instead of the typical proactive request/reply model. Thus, some of the overhead from the users requesting data is reduced and transferred to the users that store the data and can provide it.

Then, for the solution addressing scenarios *surviving without infrastructure*, the thesis discusses Thyme, a data storage and dissemination system for wireless edge environments that implements TARS. Thyme makes opportunistic use of mobile devices and ad-hoc networking to provide a transient storage service in a localized geographical region. For this, we propose two different approaches: Thyme-LS, following a lightweight unstructured approach using local storage and query flooding; and Thyme-DCS, embracing a more intricate structured approach using a storage substrate built over a cell-based geographic hash table (GHT) for wireless networks. Additionally, we implemented the Thyme-DCS approach as an Android library. The innovative characteristics of Thyme offer a novel way for sharing and accessing data that has been previously stored, or is being generated in quasi-real-time, in a network of co-located mobile devices without infrastructure access.

Regarding the solutions addressing scenarios *thriving with infrastructure*, the thesis discusses two approaches. Parsley addresses challenges in managing highly dynamic device population and workload imbalances in the context of distributed hash tables (DHTs). To tackle these issues, it relies on a resilient group-based DHT embodying two techniques. First, a preemptive peer relocation technique, enabling the transfer of individual peers between large and small groups of peers. Then, a dynamic data sharding mechanism, addressing the issue with storage hot-spots.

In turn, Thyme GardenBed is a data storage and dissemination system for multi-region edge networks. It cooperatively and symbiotically leverages both device-to-device (D2D) interactions and edge servers to allow the flow of content in networks of mobile devices spanning across multiple edge network regions. Mobile devices associated with an access point (AP) run an (adapted) instance of Thyme. In turn, edge servers run the GardenBed component and are leveraged to cache some (popular) data, and perform some of the system's management. Thus, Thyme GardenBed makes a fundamental energy shift. By introducing edge servers into the equation, it allows the optimization and offloading of a portion of the system's management and data from the clients to the stationary nodes. Ultimately, saving energy in the mobile devices and lowering their processing overheads.

Finally, considering the work presented in the thesis and its core contributions, a positive answer can be provided to the fundamental question addressed by the thesis: *How to support resilient and efficient data storage and dissemination solutions in pervasive edge computing environments, operating with or without access to network infrastructure?*

First, by developing Thyme, a data storage and dissemination system for wireless edge environments, the work presented in §4 answers our first sub-question: *How to support reliable and efficient data storage and dissemination in wireless edge environments without access to any kind of network infrastructure?* As discussed in 4.10, Thyme is reliable since it proved to be able to handle a considerable amount of churn while performing without data loss. It is also efficient since it proved to have a good performance in its target environment, while providing low energy consumption.

Next, by developing Parsley, a structured overlay with a special focus on load balancing, and by Thyme GardenBed symbiotically integrating edge resources with our previous system, the works reported in §5 and §6, respectively, answer the second sub-question: *How to leverage on edge computing capabilities to improve the performance, scalability, and resource management of the previous solution?* As discussed in 5.6, Parsley and its techniques demonstrated to improve the fault tolerance, scalability, and load balancing of group-based DHTs. The first, by achieving high availability in face of (intensive) churn, through its group-based techniques. The second, by requiring smaller bandwidth costs when executing topology changes, through its preemptive peer relocation (PPR) technique. And the last, by promoting good storage load balancing in the presence of skewed data, through its dynamic data sharding mechanism. In turn, as discussed in 6.7, Thyme GardenBed allows the flow of content in networks of mobile devices spanning across edge network regions, and showed to improve over Thyme regarding the three defined vectors: performance, scalability, and resource management. Here, all the three vectors are accomplished through the shift of some of the system management from the mobile clients to the edge servers, and their balanced integration. Thus, working to achieve low response times allowing interactive usage, low energy consumption, and considerable latency speedups over cloud solutions.

Additionally, the works described in §7 complement the contributions of the thesis by further exploiting some related research directions, from distributed computing to data synchronization techniques, and even multiplayer games. In the end, they showcase a plethora of use cases demonstrating the flexibility and usefulness of the thesis' proposals.

In the end, the results presented and discussed throughout the thesis have studied different solutions addressing data storage and dissemination in pervasive edge computing environments, either with or without access to network infrastructures. As discussed throughout the previous chapters, all approaches were also able to address the defined set of broad challenges described in §1.3. In sum, we claim that the achieved contributions are indisputable and demonstrate the validity of out thesis: *It is possible to provide resilient and efficient data storage and dissemination solutions for pervasive edge computing environments, able to operate with or without access to network infrastructure.*

8.2 Future Research Directions

Naturally, a final note resulting from the thesis is that there are still multiple open research directions that can be pursued in the field of edge computing, and more specifically for data storage and dissemination in these environments. Next, we discuss potential directions for the outcomes of this work and the research area in general.

Integration with the Cloud. An interesting research direction is to explore the next level of the network hierarchy—the cloud. With Thyme, we provide data storage and dissemination in ad-hoc settings. Then, Parsley and Thyme GardenBed bridge the gap to edge resources. A possible next step can be to yet again bridge the other gap to cloud resources/services—the actual edge-cloud continuum.

In this context, edge servers (and possibly client devices) can connect to the cloud, exploring the more robust guarantees available there. For instance, cloud services can be used for storage/archival, analytics purposes, or some kind of heavy data processing. In this solution, data (and its replicas) could be scattered throughout the network (perhaps hierarchically structured), thus greatly increasing data availability and reliability.

In the end, this would allow the deployment of something like a "planetary" data storage and dissemination system, fully exploiting the resources and guarantees available at each level of the network hierarchy. Additionally, inside this solution, several other challenges could be addressed, such as scalability and load balancing, heterogeneity, or partitions and limited connectivity.

Currently, we have a Master student that is exploring this topic in the context of the Edge Garden ecosystem (§7.9).

Client Mobility. Another possible research direction is leveraging user mobility in these scenarios. The provision of uninterrupted edge services to a frequently "on the move" client is a big challenge in edge computing environments, requiring transparent processing and/or data migration.

In fact, in this kind of dynamic environments, client mobility can be explored to propagate data among different edge regions, as in opportunistic and delay tolerant networks. Usually, mobility can be tracked either by monitoring the client's wireless connection signal strength, or by trying to build a prediction model for the client mobility pattern. Also, in this case, special attention needs to be taken regarding data consistency.

Data Abstractions and Consistency. Other relevant research direction concerns data storage abstractions and consistency semantics. Applications deployed on edge environments usually deal with different data types (e.g., images, video, text), and require different consistency semantics. In fact, sometimes the same application may require several consistency semantics for different types of data it manages.

Many proposed edge storage solutions use the well-known key-value interface for its simplicity. Nonetheless, having an expressive storage application programming interface (API) can be beneficial to run common operations (in some specific areas). Thus, domain-specific data abstractions have the potential to make edge applications more efficient, and are likely to reduce network traffic.

In turn, the majority of the proposed edge storage systems support only a single consistency semantics. However, applications typically have different consistency requirements, either contextual, location-based, or state-dependent. Thus, a multi-consistency semantics (with an easy-to-use policy interface) would enable edge applications to dynamically trade-off performance with consistency, to match the dynamic nature of their corresponding data sets and use cases. For instance, Pileus [267] is a cloud storage system that allows consistency-based service level agreements to be defined over each issued operation to the data store.

Infrastructure and Infrastructure-less Transition. Another different possibility is to explore the seamless transition between infrastructure and infrastructure-less scenarios. Currently, Thyme works in both settings (with the help of Thyme GardenBed in the case of infrastructure settings). However, it must be configured for each setting before the system bootstraps. Thus, it is unable to start working in ad-hoc mode and transition to infrastructure mode. Something like a somewhat seamless transition between modes is not trivial at all, but would increase even more the ubiquity and versatility of the system. It would probably require some mapping between the two overlay approaches, and some mechanism to handle both types of metadata.

Wireless Technologies. In a more practical aspect, the study of recent wireless technologies is also pertinent [219, 265]. Wireless communication technologies and APIs, like Wi-Fi Direct, Wi-Fi Aware, or Google Nearby, can likely reduce the amount of network traffic required to go through the APs, and allow devices to communicate in a truly D2D way. In an extreme case, these technologies can reduce, or even eliminate, the need for network infrastructures.

Bibliography

[1] ZTE Corporation. URL: https://www.zte.com.cn/global/ (cit. on p. 166).

[2] Amazon Web Services. URL: http://aws.amazon.com (cit. on p. 167).

[3] M. Afanasyev, T. Chen, G. M. Voelker, and A. C. Snoeren. "Usage Patterns in an Urban WiFi Network". In: *IEEE/ACM Trans. Netw.* 18.5 (2010), pp. 1359–1372. DOI: 10.1109/TNET.2010.2040087. URL: https://doi.org/10.1109/TNET.2010.2040087 (cit. on p. 2).

[4] J. Afonso. "Key-Value Storage for Handling Data in Mobile Devices". http://hdl.handle.net/10362/92282. MA thesis. NOVA University Lisbon, Dec. 2019 (cit. on p. 183).

[5] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. "Matching Events in a Content-based Subscription System". In: *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '99. Atlanta, Georgia, USA: ACM, 1999, pp. 53–61. ISBN: 1-58113-099-6. DOI: 10.1145/301308.301326. URL: http://doi.acm.org/10.1145/301308.301326 (cit. on pp. 22, 48, 66).

[6] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. "A survey of information-centric networking". In: *IEEE Communications Magazine* 50.7 (July 2012), pp. 26–36. ISSN: 0163-6804. DOI: 10.1109/MCOM.2012.6231276. URL: https://doi.org/10.1109/MCOM.2012.6231276 (cit. on pp. 32, 39).

[7] S. Ahuja, N. Carriero, and D. Gelernter. "Linda and Friends". In: *Computer* 19.8 (Aug. 1986), pp. 26–34. ISSN: 0018-9162. DOI: 10.1109/MC.1986.1663305. URL: http://dx.doi.org/10.1109/MC.1986.1663305 (cit. on pp. 30, 40).

[8] I. F. Akyildiz, X. Wang, and W. Wang. "Wireless mesh networks: a survey". In: *Computer Networks* 47.4 (2005), pp. 445–487. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2004.12.001. URL: https://doi.org/10.1016/j.comnet.2004.12.001 (cit. on p. 16).

[9] J. Albadarneh, Y. Jararweh, M. Al-Ayyoub, M. Al-Smadi, and R. Fontes. "Software Defined Storage for cooperative Mobile Edge Computing systems". In: *Fourth International Conference on Software Defined Systems*. SDS '17. Valencia, Spain: IEEE, 2017, pp. 174–179 (cit. on pp. 151, 152).

[10] L. Allen, A. O'Connell, and V. Kiermer. "How can we ensure visibility and diversity in research contributions? How the Contributor Role Taxonomy (CRediT) is helping the shift from authorship to contributorship". In: *Learn. Publ.* 32.1 (2019), pp. 71–74. DOI: 10.1002/leap.1210. URL: https://doi.org/10.1002/leap.1210 (cit. on p. 155).

[11] M. Amadeo, A. Molinaro, and G. Ruggeri. "E-CHANET: Routing, Forwarding and Transport in Information-Centric Multihop Wireless Networks". In: *Comput. Commun.* 36.7 (Apr. 2013), pp. 792–803. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2013.01.006. URL: http://dx.doi.org/10.1016/j.comcom.2013.01.006 (cit. on p. 39).

[12] C. Anastasiades, A. Sittampalam, and T. Braun. "Content Discovery in Wireless Information-centric Networks". In: *Proceedings of the 2015 IEEE 40th Conference on Local Computer Networks*. LCN '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 28–36. ISBN: 978-1-4673-6770-7. DOI: 10.1109/LCN.2015.7366280. URL: http://dx.doi.org/10.1109/LCN.2015.7366280 (cit. on p. 39).

[13] D. P. Anderson. "BOINC: A System for Public-Resource Computing and Storage". In: *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10. ISBN: 0-7695-2256-4. DOI: 10.1109/GRID.2004.14. URL: http://dx.doi.org/10.1109/GRID.2004.14 (cit. on p. 54).

[14] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. "SETI@Home: An Experiment in Public-resource Computing". In: *Commun. ACM* 45.11 (Nov. 2002), pp. 56–61. ISSN: 0001-0782. DOI: 10.1145/581571.581573. URL: http://doi.acm.org/10.1145/581571.581573 (cit. on p. 54).

[15] D. P. Anderson and G. Fedak. "The Computational and Storage Potential of Volunteer Computing". In: *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*. CCGRID '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 73–80. ISBN: 0-7695-2585-7. DOI: 10.1109/CCGRID.2006.101. URL: http://dx.doi.org/10.1109/CCGRID.2006.101 (cit. on p. 54).

[16] Anmobi, Inc. *Xender*. http://www.xender.com/. Accessed: 2018-05-07. 2014 (cit. on p. 63).

[17] F. Araujo, L. Rodrigues, J. Kaiser, C. Liu, and C. Mitidieri. "CHR: A Distributed Hash Table for Wireless Ad Hoc Networks". In: *Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS)*. ICDCSW '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 407–413. ISBN: 0-7695-2328-5-04. DOI: 10.1109/ICDCSW.2005.48. URL: http://dx.doi.org/10.1109/ICDCSW.2005.48 (cit. on pp. 59–61, 65, 72, 73).

[18] 5. I. Association. *5G Vision - The 5G Infrastructure Public Private Partnership: The next generation of communication networks and services*. Tech. rep. 5G Infrastructure Association, Feb. 2015 (cit. on p. 4).

[19] M. Astley, J. Auerbach, S. Bhola, G. Buttner, M. Kaplan, K. Miller, R. Saccone Jr, R. Strom, D. C. Sturman, M. J. Ward, et al. *Achieving scalability and throughput in a publish/subscribe system*. Tech. rep. IBM Research, 2004 (cit. on pp. 22, 24).

[20] S. Babu. "Continuous Query". In: *Encyclopedia of Database Systems*. Ed. by L. LIU and M. T. ÖZSU. Boston, MA: Springer US, 2009, pp. 492–493. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_85. URL: https://doi.org/10.1007/978-0-387-39940-9_85 (cit. on p. 46).

[21] S. Babu and J. Widom. "Continuous Queries over Data Streams". In: *SIGMOD Rec.* 30.3 (2001), pp. 109–120. DOI: 10.1145/603867.603884. URL: https://doi.org/10.1145/603867.603884 (cit. on p. 47).

[22] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. "Generic Support for Distributed Applications". In: *Computer* 33.3 (Mar. 2000), pp. 68–76. ISSN: 0018-9162. DOI: 10.1109/2.825698. URL: http://dx.doi.org/10.1109/2.825698 (cit. on p. 25).

[23] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems". In: *Proceedings of the 19th International Conference on Distributed Computing Systems*. ICDCS '99. Austin, Texas, USA: IEEE Computer Society, 1999, pp. 262–272. DOI: 10.1109/ICDCS.1999.776528. URL: https://doi.org/10.1109/ICDCS.1999.776528 (cit. on p. 66).

[24] D. Barbará. "The Characterization of Continuous Queries". In: *Int. J. Cooperative Inf. Syst.* 8.4 (1999), p. 295. DOI: 10.1142/S0218843099000150. URL: https://doi.org/10.1142/S0218843099000150 (cit. on p. 47).

[25] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. "Consistent Streaming Through Time: A Vision for Event Stream Processing". In: *Third Biennial Conference on Innovative Data Systems Research*. CIDR '07. Asilomar, California, USA: www.cidrdb.org, 2007, pp. 363–374. URL: http://cidrdb.org/cidr2007/papers/cidr07p42.pdf (cit. on p. 29).

[26] A. Barreto. "Conflict-Free Replicated Data Types in Dynamic Environments". http://hdl.handle.net/10362/93770. MA thesis. NOVA University Lisbon, Dec. 2019 (cit. on p. 184).

[27] A. Barreto, J. A. Silva, H. Paulino, and N. Preguiça. "CRDTs em Ambientes Dinâmicos". In: *Proceedings of the 11th Simpósio Nacional de Informática*. INForum '19. Guimarães, Portugal, 2019 (cit. on p. 185).

[28] S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic. *Mobile ad hoc networking*. John Wiley & Sons, 2004. DOI: 10.1002/0471656895 (cit. on p. 16).

[29] S. Bazarbayev, M. Hiltunen, K. Joshi, W. H. Sanders, and R. Schlichting. "PSCloud: A Durable Context-aware Personal Storage Cloud". In: *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*. HotDep '13. Farmington, Pennsylvania: ACM, 2013, 9:1–9:6. ISBN: 978-1-4503-2457-1. DOI: 10.1145/2524224.2524235. URL: http://doi.acm.org/10.1145/2524224.2524235 (cit. on p. 31).

[30] J. Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: *CoRR* abs/1407.3561 (2014). arXiv: 1407.3561. URL: http://arxiv.org/abs/1407.3561 (cit. on p. 104).

[31] A. N. Bessani, M. Correia, J. da Silva Fraga, and L. C. Lung. "An Efficient Byzantine-Resilient Tuple Space". In: *IEEE Trans. Computers* 58.8 (2009), pp. 1080–1094. DOI: 10.1109/TC.2009.71. URL: https://doi.org/10.1109/TC.2009.71 (cit. on p. 40).

[32] A. N. Bessani, J. Sousa, and E. A. P. Alchieri. "State Machine Replication for the Masses with BFT-SMART". In: *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '14. Atlanta, GA, USA: IEEE Computer Society, 2014, pp. 355–362. DOI: 10.1109/DSN.2014.43. URL: https://doi.org/10.1109/DSN.2014.43 (cit. on p. 79).

[33] S. Bhola, Y. Zhao, and J. Auerbach. "Scalably supporting durable subscriptions in a publish/subscribe system". In: *Proceedings of the International Conference on Dependable Systems and Networks*. DSN '03. IEEE Computer Society, June 2003, pp. 57–66. DOI: 10.1109/DSN.2003.1209916. URL: https://doi.org/10.1109/DSN.2003.1209916 (cit. on p. 28).

[34] K. P. Birman. "The Process Group Approach to Reliable Distributed Computing". In: *Commun. ACM* 36.12 (Dec. 1993), pp. 37–53. ISSN: 0001-0782. DOI: 10.1145/163298.163303. URL: http://doi.acm.org/10.1145/163298.163303 (cit. on p. 21).

[35] C. Blake and R. Rodrigues. "High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two". In: *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*. HotOS '03. Lihue (Kauai), Hawaii, USA, 2003, pp. 1–6. URL: https://www.usenix.org/conference/hotos-ix/high-availability-scalable-storage-dynamic-peer-networks-pick-two (cit. on p. 38).

[36] C. Boldrini, M. Conti, and A. Passarella. "ContentPlace: Social-aware Data Dissemination in Opportunistic Networks". In: *Proceedings of the 11th International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. MSWiM '08. Vancouver, British Columbia, Canada: ACM, 2008, pp. 203–210. ISBN: 978-1-60558-235-1. DOI: 10.1145/1454503.1454541. URL: http://doi.acm.org/10.1145/1454503.1454541 (cit. on p. 32).

[37] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the Workshop on Mobile Cloud Computing*. MCC '12. Helsinki, Finland: ACM, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. URL: http://doi.acm.org/10.1145/2342509.2342513 (cit. on p. 3).

[38] Briar Project. *Briar*. https://briarproject.org/. Accessed: 2018-04-27. 2017 (cit. on p. 63).

[39] F. Cao and J. P. Singh. "MEDYM: Match-early with Dynamic Multicast for Content-based Publish-subscribe Networks". In: *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*. Middleware '05. Grenoble, France: Springer-Verlag New York, Inc., 2005, pp. 292–313. URL: http://dl.acm.org/citation.cfm?id=1515890.1515905 (cit. on p. 25).

[40] M. Caporuscio, A. Carzaniga, and A. L. Wolf. "Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications". In: *IEEE Trans. Softw. Eng.* 29.12 (Dec. 2003), pp. 1059–1071. ISSN: 0098-5589. DOI: 10.1109/TSE.2003.1265521. URL: http://dx.doi.org/10.1109/TSE.2003.1265521 (cit. on p. 27).

[41] N. Carvalho, F. Araujo, and L. Rodrigues. "Reducing Latency in Rendezvous-Based Publish-Subscribe Systems for Wireless Ad Hoc Networks". In: *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems Workshops*. ICDCSW '06. Washington, DC, USA: IEEE Computer Society, 2006. ISBN: 0-7695-2541-5. DOI: 10.1109/ICDCSW.2006.89. URL: http://dx.doi.org/10.1109/ICDCSW.2006.89 (cit. on pp. 28, 60).

[42] A. Carzaniga. "Architectures for an Event Notification Service Scalable to Wide-area Networks". PhD thesis. Milano, Italy: Politecnico di Milano, Dec. 1998. URL: http://www.inf.usi.ch/carzaniga/papers/ (cit. on pp. 22, 24, 26).

[43] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. "Achieving Scalability and Expressiveness in an Internet-scale Event Notification Service". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: ACM, 2000, pp. 219–227. ISBN: 1-58113-183-6. DOI: 10.1145/343477.343622. URL: http://doi.acm.org/10.1145/343477.343622 (cit. on p. 22).

[44] A. Carzaniga and A. L. Wolf. "Content-Based Networking: A New Communication Infrastructure". In: *Developing an Infrastructure for Mobile and Wireless Systems*. Ed. by B. König-Ries, K. Makki, S. A. M. Makki, N. Pissinou, and P. Scheuermann. Vol. 2538. Lecture Notes in Computer Science. Scottsdale, AZ, USA: Springer, 2001, pp. 59–68. DOI: 10.1007/3-540-36257-6_6. URL: https://doi.org/10.1007/3-540-36257-6%5C_6 (cit. on pp. 39, 66).

[45] M. Castro, P. Druschel, A. M. Kermarrec, and A. I. T. Rowstron. “Scribe: a large-scale and decentralized application-level multicast infrastructure”. In: *IEEE Journal on Selected Areas in Communications* 20.8 (Oct. 2002), pp. 1489–1499. ISSN: 0733-8716. DOI: 10.1109/JSAC.2002.803069. URL: https://doi.org/10.1109/JSAC.2002.803069 (cit. on pp. 21, 25, 27, 70, 104).

[46] F. Cerqueira. “Um Sistema Publicador/Subscritor com Persistência de Dados para Redes de Dispositivos Móveis”. http://hdl.handle.net/10362/28553. MA thesis. NOVA University Lisbon, Nov. 2017 (cit. on pp. 57, 59, 75, 92, 94–98).

[47] F. Cerqueira, J. A. Silva, J. 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, USA: ACM, 2017, pp. 1–6. ISBN: 978-1-4503-5171-3. DOI: 10.1145/3152360.3152362. URL: http://doi.acm.org/10.1145/3152360.3152362 (cit. on pp. 101, 166).

[48] F. Cerqueira, J. A. Silva, J. M. Lourenço, and H. Paulino. “Um Sistema Publicador/Subscritor com Persistência de Dados para Redes de Dispositivos Móveis”. In: *Proceedings of the 9th Simpósio Nacional de Informática*. INForum ’17. Aveiro, Portugal, 2017 (cit. on p. 101).

[49] G. Chandrasekaran, N. Wang, and R. Tafazolli. “Caching on the Move: Towards D2D-based Information Centric Networking for Mobile Content Distribution”. In: *Proceedings of the 2015 IEEE 40th Conference on Local Computer Networks*. LCN ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 312–320. ISBN: 978-1-4673-6770-7. DOI: 10.1109/LCN.2015.7366325. URL: http://dx.doi.org/10.1109/LCN.2015.7366325 (cit. on p. 39).

[50] K. Chintalapudi, A. Padmanabha Iyer, and V. N. Padmanabhan. “Indoor Localization Without the Pain”. In: *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*. Chicago, Illinois, USA: ACM, 2010, pp. 173–184. ISBN: 978-1-4503-0181-7. DOI: 10.1145/1859995.1860016. URL: http://doi.acm.org/10.1145/1859995.1860016 (cit. on p. 18).

[51] I. Chlamtac, M. Conti, and J. J.-N. Liu. “Mobile ad hoc networking: imperatives and challenges”. In: *Ad Hoc Networks* 1.1 (2003), pp. 13–64. ISSN: 1570-8705. DOI: 10.1016/S1570-8705(03)00013-1. URL: https://doi.org/10.1016/S1570-8705(03)00013-1 (cit. on p. 15).

[52] H.-D. Cho, K. Chung, and T. Kim. *Benefits of the big.LITTLE Architecture*. Tech. rep. Samsung, Feb. 2012 (cit. on p. 2).

[53] G. V. Chockler, I. Keidar, and R. Vitenberg. “Group Communication Specifications: A Comprehensive Study”. In: *ACM Comput. Surv.* 33.4 (Dec. 2001), pp. 427–469. ISSN: 0360-0300. DOI: 10.1145/503112.503113. URL: http://doi.acm.org/10.1145/503112.503113 (cit. on p. 21).

[54] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. "CloneCloud: Elastic Execution Between Mobile Device and Cloud". In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, pp. 301–314. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966473. URL: http://doi.acm.org/10.1145/1966445.1966473 (cit. on p. 1).

[55] M. Cilia, L. Fiege, C. Haul, A. Zeidler, and A. P. Buchmann. "Looking into the Past: Enhancing Mobile Publish/Subscribe Middleware". In: *Proceedings of the 2Nd International Workshop on Distributed Event-based Systems*. DEBS '03. San Diego, California, USA: ACM, 2003, pp. 1–8. ISBN: 1-58113-843-1. DOI: 10.1145/966618.966631. URL: http://doi.acm.org/10.1145/966618.966631 (cit. on pp. 28, 44, 45, 60).

[56] Cisco. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021*. Tech. rep. Cisco, Feb. 2017 (cit. on pp. 2, 5, 58, 150).

[57] Cisco. *The Zettabyte Era: Trends and Analysis*. Tech. rep. Cisco, June 2017 (cit. on pp. 5, 58).

[58] Cisco. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2017–2022*. Tech. rep. Cisco, Feb. 2019 (cit. on p. 2).

[59] Cisco. *Cisco Annual Internet Report (2018–2023)*. Tech. rep. Cisco, Mar. 2020 (cit. on pp. 2, 5, 58).

[60] T. H. Clausen and P. Jacquet. "Optimized Link State Routing Protocol (OLSR)". In: *RFC* 3626 (2003), pp. 1–75. DOI: 10.17487/RFC3626. URL: https://doi.org/10.17487/RFC3626 (cit. on pp. 60, 65).

[61] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*. Ed. by J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum. ACM, 2010, pp. 143–154. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152 (cit. on p. 136).

[62] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. "TeenyLIME: Transiently Shared Tuple Space Middleware for Wireless Sensor Networks". In: *Proceedings of the International Workshop on Middleware for Sensor Networks*. MidSens '06. Melbourne, Australia: ACM, 2006, pp. 43–48. ISBN: 1-59593-424-3. DOI: 10.1145/1176866.1176874. URL: http://doi.acm.org/10.1145/1176866.1176874 (cit. on p. 40).

[63] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. "MAUI: Making Smartphones Last Longer with Code Offload". In: *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. MobiSys '10. San Francisco, California, USA: ACM, 2010, pp. 49–62.

ISBN: 978-1-60558-985-5. DOI: 10.1145/1814433.1814441. URL: http://doi.acm.org/10.1145/1814433.1814441 (cit. on p. 1).

[64] G. Cugola, E. Di Nitto, and A. Fuggetta. "The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS". In: *IEEE Trans. Softw. Eng.* 27.9 (Sept. 2001), pp. 827–850. ISSN: 0098-5589. DOI: 10.1109/32.950318. URL: http://dx.doi.org/10.1109/32.950318 (cit. on pp. 22, 24).

[65] G. Cugola and H.-A. Jacobsen. "Using Publish/Subscribe Middleware for Mobile Systems". In: *SIGMOBILE Mob. Comput. Commun. Rev.* 6.4 (Oct. 2002), pp. 25–33. ISSN: 1559-1662. DOI: 10.1145/643550.643552. URL: http://doi.acm.org/10.1145/643550.643552 (cit. on p. 27).

[66] G. Cugola and A. Margara. "Processing Flows of Information: From Data Stream to Complex Event Processing". In: *ACM Comput. Surv.* 44.3 (June 2012), pp. 1–62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677. URL: http://doi.acm.org/10.1145/2187671.2187677 (cit. on p. 29).

[67] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco. "TinyLIME: Bridging Mobile and Sensor Networks Through Middleware". In: *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*. PerCom '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 61–72. ISBN: 0-7695-2299-8. DOI: 10.1109/PERCOM.2005.48. URL: http://dx.doi.org/10.1109/PERCOM.2005.48 (cit. on p. 40).

[68] F. Dabek, M. F. Kaashoek, D. R. Karger, R. T. Morris, and I. Stoica. "Wide-Area Cooperative Storage with CFS". In: *Proceedings of the 18th ACM Symposium on Operating System Principles*. Ed. by K. Marzullo and M. Satyanarayanan. SOSP '01. Chateau Lake Louise, Banff, Alberta, Canada: ACM, 2001, pp. 202–215. DOI: 10.1145/502034.502054. URL: https://doi.org/10.1145/502034.502054 (cit. on pp. 34, 37, 104).

[69] C. Dannewitz, D. Kutscher, B. Ohlman, S. Farrell, B. Ahlgren, and H. Karl. "Network of Information (NetInf) - An information-centric networking architecture". In: *Comput. Commun.* 36.7 (2013), pp. 721–735. DOI: 10.1016/j.comcom.2013.01.009. URL: https://doi.org/10.1016/j.comcom.2013.01.009 (cit. on p. 39).

[70] DARPA. *Creating a Secure, Private Internet and Cloud at the Tactical Edge*. https://www.darpa.mil/news-events/2013-08-21. Accessed: 2020-07-10. 2013 (cit. on pp. 9, 15, 58, 181).

[71] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002. ISBN: 978-0-521-78451-1. DOI: 10.1017/CBO9780511809088. URL: https://doi.org/10.1017/CBO9780511809088 (cit. on p. 70).

[72] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 2007, pp. 205–220. DOI: 10.1145/1294261.1294281. URL: https://doi.org/10.1145/1294261.1294281 (cit. on p. 104).

[73] M. Demmer, B. Du, and E. Brewer. "TierStore: A Distributed Filesystem for Challenged Networks in Developing Regions". In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. FAST'08. San Jose, California: USENIX Association, 2008, 3:1–3:14. URL: http://dl.acm.org/citation.cfm?id=1364813.1364816 (cit. on pp. 30, 62).

[74] J. Dias. "Adaptive Replica Selection in Mobile Edge Networks". MA thesis. NOVA University Lisbon, Feb. 2021 (cit. on p. 186).

[75] V. Dimitrov and V. Koptchev. "PSIRP project – publish-subscribe internet routing paradigm: new ideas for future internet". In: *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*. Ed. by B. Rachev and A. Smrikarov. CompSysTech '10. Sofia, Bulgaria: ACM, 2010, pp. 167–171. DOI: 10.1145/1839379.1839409. URL: https://doi.org/10.1145/1839379.1839409 (cit. on p. 39).

[76] A. Doan, R. Ramakrishnan, and A. Y. Halevy. "Crowdsourcing Systems on the World-Wide Web". In: *Commun. ACM* 54.4 (Apr. 2011), pp. 86–96. ISSN: 0001-0782. DOI: 10.1145/1924421.1924442. URL: http://doi.acm.org/10.1145/1924421.1924442 (cit. on p. 54).

[77] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan. "Cachier: Edge-Caching for Recognition Applications". In: *37th IEEE International Conference on Distributed Computing Systems*. ICDCS '17. Atlanta, GA, USA: IEEE Computer Society, 2017, pp. 276–286. DOI: 10.1109/ICDCS.2017.94. URL: https://doi.org/10.1109/ICDCS.2017.94 (cit. on pp. 31, 150).

[78] U. Drolia, R. Martins, J. Tan, A. Chheda, M. Sanghavi, R. Gandhi, and P. Narasimhan. "The Case for Mobile Edge-Clouds". In: *Proceedings of the 2013 IEEE 10th International Conference on Ubiquitous Intelligence & Computing and 2013 IEEE 10th International Conference on Autonomic & Trusted Computing*. UIC-ATC '13. Vietri sul Mare, Sorrento, Italy: IEEE Computer Society, 2013, pp. 209–215. ISBN: 978-1-4799-2482-0. DOI: 10.1109/UIC-ATC.2013.94. URL: http://dx.doi.org/10.1109/UIC-ATC.2013.94 (cit. on pp. 2, 4, 6).

[79] U. Drolia, N. Mickulicz, R. Gandhi, and P. Narasimhan. "Krowd: A Key-Value Store for Crowded Venues". In: *Proceedings of the 10th International Workshop on Mobility in the Evolving Internet Architecture*. MobiArch '15. Paris, France: ACM, 2015, pp. 20–25. ISBN: 978-1-4503-3695-6. DOI: 10.1145/2795381.2795388.

URL: http://doi.acm.org/10.1145/2795381.2795388 (cit. on pp. 30, 43, 59, 61, 150).

[80] G. Einziger, R. Friedman, and B. Manes. "TinyLFU: A Highly Efficient Cache Admission Policy". In: *ACM Trans. Storage* 13.4 (2017), 35:1–35:31. DOI: 10.1145/3149371. URL: https://doi.org/10.1145/3149371 (cit. on p. 165).

[81] Ericsson. *Ericsson Mobility Report*. Tech. rep. Ericsson, Nov. 2020 (cit. on pp. 2, 5).

[82] J. Erman and K. Ramakrishnan. "Understanding the Super-sized Traffic of the Super Bowl". In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC '13. Barcelona, Spain: ACM, 2013, pp. 353–360. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504770. URL: http://doi.acm.org/10.1145/2504730.2504770 (cit. on pp. 5, 58, 64).

[83] E. Estellés-Arolas and F. G. Ladrón-De-Guevara. "Towards an Integrated Crowdsourcing Definition". In: *J. Inf. Sci.* 38.2 (Apr. 2012), pp. 189–200. ISSN: 0165-5515. DOI: 10.1177/0165551512437638. URL: http://dx.doi.org/10.1177/0165551512437638 (cit. on p. 54).

[84] P. Eugster. "Type-based Publish/Subscribe: Concepts and Experiences". In: *ACM Trans. Program. Lang. Syst.* 29.1 (Jan. 2007). ISSN: 0164-0925. DOI: 10.1145/1180475.1180481. URL: http://doi.acm.org/10.1145/1180475.1180481 (cit. on p. 22).

[85] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. "The Many Faces of Publish/Subscribe". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: http://doi.acm.org/10.1145/857076.857078 (cit. on pp. 19–21, 28, 43, 48).

[86] P. T. Eugster and R. Guerraoui. "Probabilistic Multicast". In: *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. DSN '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 313–324. ISBN: 0-7695-1597-5. URL: http://dl.acm.org/citation.cfm?id=647883.738400 (cit. on p. 27).

[87] P. T. Eugster, R. Guerraoui, and C. H. Damm. "On Objects and Events". In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '01. Tampa Bay, FL, USA: ACM, 2001, pp. 254–269. ISBN: 1-58113-335-9. DOI: 10.1145/504282.504301. URL: http://doi.acm.org/10.1145/504282.504301 (cit. on p. 22).

[88] P. T. Eugster, R. Guerraoui, and J. Sventek. "Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction". In: *Proceedings of the 14th European Conference on Object-Oriented Programming*. ECOOP '00. Sophia Antipolis and Cannes, France: Springer-Verlag, 2000, pp. 252–276. ISBN: 3-540-67660-0. URL: http://dl.acm.org/citation.cfm?id=646157.758679 (cit. on p. 22).

[89] K. Fall. “A Delay-tolerant Network Architecture for Challenged Internets”. In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’03. Karlsruhe, Germany: ACM, 2003, pp. 27–34. ISBN: 1-58113-735-4. DOI: 10.1145/863955.863960. URL: http://doi.acm.org/10.1145/863955.863960 (cit. on p. 17).

[90] U. Farooq, S. Majumdar, and E. W. Parsons. *Semi-Durable Subscriptions: A Technique to Achieve High Performance in Mobile Wireless Publish/Subscribe Systems*. Tech. rep. Carleton University, July 2003 (cit. on p. 28).

[91] N. Fernando, S. W. Loke, and W. Rahayu. “Mobile Cloud Computing: A Survey”. In: *Future Generation Computer Systems* 29.1 (Jan. 2013), pp. 84–106. ISSN: 0167-739X. DOI: 10.1016/j.future.2012.05.023. URL: http://dx.doi.org/10.1016/j.future.2012.05.023 (cit. on p. 1).

[92] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. “Supporting Mobility in Content-based Publish/Subscribe Middleware”. In: *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Middleware ’03. Rio de Janeiro, Brazil: Springer-Verlag, 2003, pp. 103–122. ISBN: 3-540-40317-5. URL: http://dl.acm.org/citation.cfm?id=1515915.1515923 (cit. on p. 27).

[93] L. Fiege, A. Zeidler, F. C. Gärtner, and S. B. Handurukande. “Dealing with Uncertainty in Mobile Publish/Subscribe Middleware”. In: *International Workshop on Middleware for Pervasive and Ad-Hoc Computing*. Rio de Janeiro, Brazil: PUC-Rio, 2003, pp. 60–67 (cit. on p. 27).

[94] M. J. Freedman, E. Freudenthal, and D. Mazières. “Democratizing Content Publication with Coral”. In: *1st Symposium on Networked Systems Design and Implementation*. Ed. by R. T. Morris and S. Savage. NSDI ’04. San Francisco, California, USA: USENIX, 2004, pp. 239–252. URL: http://www.usenix.org/events/nsdi04/tech/freedman.html (cit. on pp. 34, 104).

[95] M. Garcia, J. Rodrigues, J. Silva, E. R. B. Marques, and L. M. B. Lopes. “Ramble: Opportunistic Crowdsourcing of User-Generated Data using Mobile Edge Clouds”. In: *Fifth International Conference on Fog and Mobile Edge Computing*. FMEC ’20. Paris, France: IEEE, 2020, pp. 172–179. DOI: 10.1109/FMEC49853.2020.9144881. URL: https://doi.org/10.1109/FMEC49853.2020.9144881 (cit. on pp. 125, 153).

[96] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. “Edge-centric Computing: Vision and Challenges”. In: *SIGCOMM Comput. Commun. Rev.* 45.5 (Sept. 2015), pp. 37–42. ISSN: 0146-4833. DOI: 10.1145/2831347.2831354. URL: http://doi.acm.org/10.1145/2831347.2831354 (cit. on pp. 3–5, 58, 104).

[97] M. Gast. *802.11 wireless networks: the definitive guide*. O’Reilly, 2005 (cit. on pp. 13–15).

[98] D. Gelernter. “Generative Communication in Linda”. In: *ACM Trans. Program. Lang. Syst.* 7.1 (Jan. 1985), pp. 80–112. ISSN: 0164-0925. DOI: 10.1145/2363.2433. URL: http://doi.acm.org/10.1145/2363.2433 (cit. on pp. 30, 40).

[99] A. Ghodsi, L. O. Alima, and S. Haridi. “Symmetric Replication for Structured Peer-to-Peer Systems”. In: *Databases, Information Systems, and Peer-to-Peer Computing, International Workshops*. Ed. by G. Moro, S. Bergamaschi, S. Joseph, J.-H. Morin, and A. M. Ouksel. Vol. 4125. DBISP2P ’05. Trondheim, Norway: Springer, 2005, pp. 74–85. DOI: 10.1007/978-3-540-71661-7_7. URL: https://doi.org/10.1007/978-3-540-71661-7%5C_7 (cit. on p. 118).

[100] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. E. Anderson. “Scalable consistency in Scatter”. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal, 2011, pp. 15–28. DOI: 10.1145/2043556.2043559. URL: https://doi.org/10.1145/2043556.2043559 (cit. on p. 38).

[101] B. Godfrey, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica. “Load Balancing in Dynamic Structured P2P Systems”. In: *Proceedings of The 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*. INFOCOM ’04. Hong Kong, China, 2004, pp. 2253–2262. DOI: 10.1109/INFCOM.2004.1354648. URL: https://doi.org/10.1109/INFCOM.2004.1354648 (cit. on p. 37).

[102] goTenna, Inc. *goTenna Mesh*. https://www.gotenna.com/. Accessed: 2020-10-05. 2017 (cit. on p. 63).

[103] O. M. Group. *Data Distribution Service for Real-time Systems Version 1.2*. Tech. rep. Object Management Group, Jan. 2007 (cit. on p. 21).

[104] Guifi.net. *Guifi.net*. https://guifi.net/. Accessed: 2020-07-20. 2017 (cit. on p. 16).

[105] P. K. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. “The impact of DHT routing geometry on resilience and proximity”. In: *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Ed. by A. Feldmann, M. Zitterbart, J. Crowcroft, and D. Wetherall. SIGCOMM ’03. Karlsruhe, Germany: ACM, 2003, pp. 381–394. DOI: 10.1145/863955.863998. URL: https://doi.org/10.1145/863955.863998 (cit. on p. 34).

[106] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. “Meghdoot: Content-based Publish/Subscribe over P2P Networks”. In: *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*. Middleware ’04. Toronto, Canada: Springer-Verlag New York, Inc., 2004, pp. 254–273. ISBN: 3-540-23428-4. URL: http://dl.acm.org/citation.cfm?id=1045658.1045677 (cit. on p. 27).

[107] H. Gupta and U. Ramachandran. “FogStore: A Geo-Distributed Key-Value Store Guaranteeing Low Latency for Strongly Consistent Access”. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*. DEBS ’18. Hamilton, New Zealand: Association for Computing Machinery, 2018, pp. 148–159. ISBN: 9781450357821. DOI: 10.1145/3210284.3210297. URL: https://doi.org/10.1145/3210284.3210297 (cit. on p. 31).

[108] I. Gupta, K. P. Birman, P. Linga, A. J. Demers, and R. van Renesse. “Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead”. In: *Peer-to-Peer Systems II, Second International Workshop*. IPTPS ’03. Berkeley, California, USA, 2003, pp. 160–169. DOI: 10.1007/978-3-540-45172-3_15. URL: https://doi.org/10.1007/978-3-540-45172-3%5C_15 (cit. on p. 38).

[109] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. “Towards Wearable Cognitive Assistance”. In: *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys ’14. Bretton Woods, New Hampshire, USA: ACM, 2014, pp. 68–81. ISBN: 978-1-4503-2793-0. DOI: 10.1145/2594368.2594383. URL: http://doi.acm.org/10.1145/2594368.2594383 (cit. on p. 2).

[110] K. Ha, P. Pillai, G. Lewis, S. Simanta, S. Clinch, N. Davies, and M. Satyanarayanan. “The Impact of Mobile Multimedia Applications on Data Center Consolidation”. In: *Proceedings of the 2013 IEEE International Conference on Cloud Engineering*. IC2E ’13. San Francisco, California, USA: IEEE Computer Society, 2013, pp. 166–176. ISBN: 978-0-7695-4945-3. DOI: 10.1109/IC2E.2013.17. URL: http://dx.doi.org/10.1109/IC2E.2013.17 (cit. on pp. 2, 6, 58, 150).

[111] Z. J. Haas. “A new routing protocol for the reconfigurable wireless networks”. In: *Proceedings of the 6th International Conference on Universal Personal Communications*. Vol. 2. ICUPC ’97. Nov. 1997, 562–566 vol.2. DOI: 10.1109/ICUPC.1997.627227 (cit. on p. 18).

[112] Z. Hao and Q. Li. “Poster Abstract: EdgeStore: Integrating Edge Computing into Cloud-Based Storage Systems”. In: *IEEE/ACM Symposium on Edge Computing*. SEC ’16. Washington, DC,USA: IEEE Computer Society, 2016, pp. 115–116. DOI: 10.1109/SEC.2016.34. URL: https://doi.org/10.1109/SEC.2016.34 (cit. on p. 31).

[113] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. *Java Message Service*. Tech. rep. Sun Microsystems Inc., 2013 (cit. on pp. 21, 22, 24).

[114] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. “SkipNet: A Scalable Overlay Network with Practical Locality Properties”. In: *4th USENIX Symposium on Internet Technologies and Systems*. Ed. by S. D. Gribble. USITS

'03. Seattle, Washington, USA: USENIX, 2003. URL: http://www.usenix.org/events/usits03/tech/harvey.html (cit. on p. 34).

[115] S. Hosio, D. Ferreira, J. Gonçalves, N. van Berkel, C. Luo, M. Ahmed, H. Flores, and V. Kostakos. "Monetary Assessment of Battery Life on Smartphones". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. Ed. by J. Kaye, A. Druin, C. Lampe, D. Morris, and J. P. Hourcade. CHI '16. San Jose, CA, USA: ACM, 2016, pp. 1869–1880. DOI: 10.1145/2858036.2858285. URL: https://doi.org/10.1145/2858036.2858285 (cit. on p. 2).

[116] J. Howe. "The rise of crowdsourcing". In: *Wired magazine* 14.6 (2006), pp. 1–4 (cit. on p. 54).

[117] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan. "Quantifying the Impact of Edge Computing on Mobile Applications". In: *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '16. Hong Kong, Hong Kong: ACM, 2016, 5:1–5:8. ISBN: 978-1-4503-4265-0. DOI: 10.1145/2967360.2967369. URL: http://doi.acm.org/10.1145/2967360.2967369 (cit. on pp. 2, 6, 58, 150).

[118] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. *Mobile Edge Computing - A key technology towards 5G*. Tech. rep. European Telecommunications Standards Institute, Sept. 2015 (cit. on p. 4).

[119] Huawei. *Huawei P30*. https://consumer.huawei.com/en/phones/p30/. Accessed: 220-09-23. 2020 (cit. on p. 2).

[120] M. Ionescu and I. Marsic. "Stateful Publish-subscribe for Mobile Environments". In: *Proceedings of the 2Nd ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots*. WMASH '04. Philadelphia, PA, USA: ACM, 2004, pp. 21–28. ISBN: 1-58113-877-6. DOI: 10.1145/1024733.1024737. URL: http://doi.acm.org/10.1145/1024733.1024737 (cit. on p. 28).

[121] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. Goren, and C. Mahmoudi. *The NIST Definition of Fog Computing*. Tech. rep. National Institute of Standards and Technology, Aug. 2017 (cit. on p. 3).

[122] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. Braynard. "Networking named content". In: *Proceedings of the 2009 ACM Conference on Emerging Networking Experiments and Technology*. Ed. by J. Liebeherr, G. Ventre, E. W. Biersack, and S. Keshav. CoNEXT '09. Rome, Italy: ACM, 2009, pp. 1–12. DOI: 10.1145/1658939.1658941. URL: https://doi.org/10.1145/1658939.1658941 (cit. on p. 39).

[123] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. "Optimized link state routing protocol for ad hoc networks". In: *Proceedings of the IEEE International Multi Topic Conference*. IEEE INMIC 2001. 2001, pp. 62–68. DOI: 10.1109/INMIC.2001.995315 (cit. on p. 17).

[124] S. Jain, K. Fall, and R. Patra. “Routing in a Delay Tolerant Network”. In: *SIGCOMM Comput. Commun. Rev.* 34.4 (Aug. 2004), pp. 145–158. ISSN: 0146-4833. DOI: 10.1145/1030194.1015484. URL: http://doi.acm.org/10.1145/1030194.1015484 (cit. on p. 17).

[125] A. P. Jardosh, K. N. Ramachandran, K. C. Almeroth, and E. M. Belding-Royer. “Understanding Congestion in IEEE 802.11b Wireless Networks”. In: *Proceedings of the 5th Internet Measurement Conference*. IMC ’05. Berkeley, California, USA: USENIX Association, 2005, pp. 279–292. URL: http://www.usenix.org/events/imc05/tech/jardosh.html (cit. on p. 2).

[126] Y. Jiang, G. Xue, Z. Jia, and J. You. “DTuples: A Distributed Hash Table based Tuple Space Service for Distributed Coordination”. In: *Grid and Cooperative Computing - GCC 2006, 5th International Conference, Changsha, Hunan, China, 21-23 October 2006, Proceedings*. IEEE Computer Society, 2006, pp. 101–106. DOI: 10.1109/GCC.2006.41. URL: https://doi.org/10.1109/GCC.2006.41 (cit. on p. 40).

[127] Y. Jiang, X. Xu, P. Terlecky, T. Abdelzaher, A. Bar-Noy, and R. Govindan. “MediaScope: Selective On-demand Media Retrieval from Mobile Devices”. In: *Proceedings of the 12th International Conference on Information Processing in Sensor Networks*. IPSN ’13. Philadelphia, Pennsylvania, USA: ACM, 2013, pp. 289–300. ISBN: 978-1-4503-1959-1. DOI: 10.1145/2461381.2461416. URL: http://doi.acm.org/10.1145/2461381.2461416 (cit. on p. 32).

[128] D. Johnson, Y. Hu, and D. Maltz. *The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4*. RFC 4728. http://www.rfc-editor.org/rfc/rfc4728.txt. RFC Editor, Feb. 2007. URL: http://www.rfc-editor.org/rfc/rfc4728.txt (cit. on p. 18).

[129] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. “Energy-efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet”. In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. San Jose, California, USA: ACM, 2002, pp. 96–107. ISBN: 1-58113-574-2. DOI: 10.1145/605397.605408. URL: http://doi.acm.org/10.1145/605397.605408 (cit. on pp. 16, 17).

[130] W. S. Jung, H. Ahn, and Y. B. Ko. “Designing content-centric multi-hop networking over Wi-Fi Direct on smartphones”. In: *IEEE Wireless Communications and Networking Conference*. WCNC ’14. Istanbul, Turkey: IEEE, Apr. 2014, pp. 2934–2939. DOI: 10.1109/WCNC.2014.6952920. URL: https://doi.org/10.1109/WCNC.2014.6952920 (cit. on p. 39).

[131] M. Kamel, C. M. Scoglio, and T. Easton. "Optimal Topology Design for Overlay Networks". In: *6th International IFIP Networking Conference*. Ed. by I. F. Akyildiz, R. Sivakumar, E. Ekici, J. C. de Oliveira, and J. McNair. Vol. 4479. Lecture Notes in Computer Science. Atlanta, GA, USA: Springer, 2007, pp. 714–725. DOI: 10.1007/978-3-540-72606-7_61. URL: https://doi.org/10.1007/978-3-540-72606-7%5C_61 (cit. on p. 33).

[132] P.-H. Kamp. "LinkedIn Password Leak: Salt Their Hide". In: *ACM Queue* 10.6 (2012), p. 20. DOI: 10.1145/2246036.2254400. URL: https://doi.org/10.1145/2246036.2254400 (cit. on p. 120).

[133] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, , May 4-6, 1997*. STOC '97. El Paso, Texas, USA, 1997, pp. 654–663. DOI: 10.1145/258533.258660. URL: https://doi.org/10.1145/258533.258660 (cit. on pp. 34, 104).

[134] D. R. Karger and M. Ruhl. "Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems". In: *Peer-to-Peer Systems III, Third International Workshop*. IPTPS '04. La Jolla, California, USA, 2004, pp. 131–140. DOI: 10.1007/978-3-540-30183-7_13. URL: https://doi.org/10.1007/978-3-540-30183-7%5C_13 (cit. on p. 37).

[135] B. Karp and H. T. Kung. "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks". In: *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*. MobiCom '00. Boston, Massachusetts, USA: ACM, 2000, pp. 243–254. ISBN: 1-58113-197-6. DOI: 10.1145/345910.345953. URL: http://doi.acm.org/10.1145/345910.345953 (cit. on pp. 18, 72, 77).

[136] K. Kenthapadi and G. S. Manku. "Decentralized algorithms using both local and random probes for P2P load balancing". In: *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '05. Las Vegas, Nevada, USA, 2005, pp. 135–144. DOI: 10.1145/1073970.1073990. URL: https://doi.org/10.1145/1073970.1073990 (cit. on p. 37).

[137] A. R. Khakpour and I. Demeure. "Chapar: A Persistent Overlay Event System for MANETs". In: *Mob. Netw. Appl.* 15.6 (Dec. 2010), pp. 866–875. ISSN: 1383-469X. DOI: 10.1007/s11036-010-0238-6. URL: http://dx.doi.org/10.1007/s11036-010-0238-6 (cit. on pp. 29, 45, 60).

[138] A. Khan, M. Attique, Y. Kim, S. Park, and B.-C. Tak. "EDGESTORE: A Single Namespace and Resource-Aware Federation File System for Edge Servers". In: *2018 IEEE International Conference on Edge Computing*. EDGE '18. San Francisco, California, USA: IEEE Computer Society, 2018, pp. 101–108. DOI: 10.1109/EDGE.2018.00021. URL: https://doi.org/10.1109/EDGE.2018.00021 (cit. on p. 31).

[139] M. A. Khan, L. Yeh, K. Zeitouni, and C. Borcea. “MobiStore: A system for efficient mobile P2P data sharing”. In: *Peer-to-Peer Networking and Applications* 10.4 (2017), pp. 910–924. DOI: 10.1007/s12083-016-0450-7. URL: https://doi.org/10.1007/s12083-016-0450-7 (cit. on pp. 38, 105, 124, 144).

[140] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. “A data-oriented (and beyond) network architecture”. In: *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Ed. by J. Murai and K. Cho. SIGCOMM ’07. Kyoto, Japan: ACM, 2007, pp. 181–192. DOI: 10.1145/1282380.1282402. URL: https://doi.org/10.1145/1282380.1282402 (cit. on p. 39).

[141] J. Kreps, N. Narkhede, J. Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *NetDB*. 2011, pp. 1–7 (cit. on pp. 44, 60).

[142] A. Krifa, M. K. Sbai, C. Barakat, and T. Turletti. “BitHoc: A Content Sharing Application for Wireless Ad Hoc Networks”. In: *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications*. PERCOM ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–3. ISBN: temp-isbn. DOI: 10.1109/PERCOM.2009.4912792. URL: https://doi.org/10.1109/PERCOM.2009.4912792 (cit. on p. 33).

[143] S. Ktari, M. Zoubert, A. Hecker, and H. Labiod. “Performance evaluation of replication strategies in DHTs under churn”. In: *Proceedings of the 6th International Conference on Mobile and Ubiquitous Multimedia*. MUM ’07. Oulu, Finland, 2007, pp. 90–97. DOI: 10.1145/1329469.1329481. URL: https://doi.org/10.1145/1329469.1329481 (cit. on p. 37).

[144] K. Kutzner and T. Fuhrmann. “Measuring Large Overlay Networks - The Overnet Example”. In: *Kommunikation in Verteilten Systemen (KiVS), 14. ITG/GI-Fachtagung Kommunikation in Verteilten Systemen*. KiVS ’05. Kaiserslautern, Germany, 2005, pp. 193–204. DOI: 10.1007/3-540-27301-8_16. URL: https://doi.org/10.1007/3-540-27301-8%5C_16 (cit. on pp. 36, 104, 105).

[145] A. Lakshman and P. Malik. “Cassandra: a decentralized structured storage system”. In: *ACM SIGOPS Oper. Syst. Rev.* 44.2 (2010), pp. 35–40. DOI: 10.1145/1773912.1773922. URL: https://doi.org/10.1145/1773912.1773922 (cit. on pp. 34, 104, 184).

[146] L. Lamport. “The Part-Time Parliament”. In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169. DOI: 10.1145/279227.279229. URL: https://doi.org/10.1145/279227.279229 (cit. on p. 38).

[147] J. Ledlie and M. I. Seltzer. “Distributed, secure load balancing with skew, heterogeneity and churn”. In: *24th Annual Joint Conference of the IEEE Computer and Communications Societies*. INFOCOM ’05. Miami, Florida, USA, 2005, pp. 1419–

1430. DOI: 10.1109/INFCOM.2005.1498366. URL: https://doi.org/10.1109/INFCOM.2005.1498366 (cit. on p. 37).

[148] T. J. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman. "Hitting the distributed computing sweet spot with TSpaces". In: *Comput. Networks* 35.4 (2001), pp. 457–472. DOI: 10.1016/S1389-1286(00)00178-X. URL: https://doi.org/10.1016/S1389-1286(00)00178-X (cit. on p. 40).

[149] J. Leitão, J. Pereira, and L. E. T. Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast". In: *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*. 2007, pp. 419–429. DOI: 10.1109/DSN.2007.56. URL: https://doi.org/10.1109/DSN.2007.56 (cit. on pp. 108, 117).

[150] J. Leitão and L. Rodrigues. "Overnesia: A Resilient Overlay Network for Virtual Super-Peers". In: *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. SRDS '14. Nara, Japan: IEEE Computer Society, 2014, pp. 281–290. ISBN: 978-1-4799-5584-8. DOI: 10.1109/SRDS.2014.40. URL: http://dx.doi.org/10.1109/SRDS.2014.40 (cit. on pp. 66, 72, 111, 124, 125).

[151] H. K. Y. Leung. "Subject Space: A State-persistent Model for Publish/Subscribe Systems". In: *Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '02. Toronto, Ontario, Canada: IBM Press, 2002. URL: http://dl.acm.org/citation.cfm?id=782115.782122 (cit. on p. 28).

[152] H. K. Y. Leung and H.-A. Jacobsen. "Efficient Matching for State-persistent Publish/Subscribe Systems". In: *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '03. Toronto, Ontario, Canada: IBM Press, 2003, pp. 182–196. URL: http://dl.acm.org/citation.cfm?id=961322.961352 (cit. on p. 28).

[153] G. Li, A. Cheung, S. Hou, S. Hu, V. Muthusamy, R. Sherafat, A. Wun, H.-A. Jacobsen, and S. Manovski. "Historic Data Access in Publish/Subscribe". In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*. DEBS '07. Toronto, Ontario, Canada: ACM, 2007, pp. 80–84. ISBN: 978-1-59593-665-3. DOI: 10.1145/1266894.1266908. URL: http://doi.acm.org/10.1145/1266894.1266908 (cit. on p. 28).

[154] Light Reading. *Edge Computing: AT&T's Next Big Play?* https://goo.gl/yLP8j5. Accessed: 2020-05-05. 2017 (cit. on p. 4).

[155] L. Liu and M. T. Zsu. *Encyclopedia of Database Systems*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 0387355448, 9780387355443 (cit. on pp. 21, 23, 28, 29).

[156] LiveQoS. *SuperBeam*. https://superbe.am/. Accessed: 2018-05-07. 2017 (cit. on p. 63).

[157] I. M. Lombera, L. E. Moser, P. M. Melliar-Smith, and Y.-T. Chuang. "Mobile ad-hoc search and retrieval in the iTrust over Wi-Fi Direct network". In: *Proc. 9th Intl. Conference on Wireless and Mobile Communications*. 2013, pp. 251–258 (cit. on pp. 32, 61).

[158] N. Lopes and C. Baquero. "Taming Hot-Spots in DHT Inverted Indexes". In: *ACM SIGIR Workshop on Large Scale Distributed Systems for Information Retrieval*. 2007 (cit. on p. 105).

[159] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. viii).

[160] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. "A survey and comparison of peer-to-peer overlay network schemes". In: *IEEE Commun. Surv. Tutorials* 7.1-4 (2005), pp. 72–93. DOI: 10.1109/COMST.2005.1610546. URL: https://doi.org/10.1109/COMST.2005.1610546 (cit. on p. 33).

[161] D. Luckham. *The power of events*. Vol. 204. Addison-Wesley Reading, 2002 (cit. on p. 29).

[162] J. Luo, J.-P. Hubaux, and P. T. Eugster. "PAN: Providing Reliable Storage in Mobile Ad Hoc Networks with Probabilistic Quorum Systems". In: *Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking &Amp; Computing*. MobiHoc '03. Annapolis, Maryland, USA: ACM, 2003, pp. 1–12. ISBN: 1-58113-684-6. DOI: 10.1145/778415.778417. URL: http://doi.acm.org/10.1145/778415.778417 (cit. on pp. 30, 43, 61).

[163] S. Luo, Z. Wen, X. Zhang, W. Xu, A. Y. Zomaya, and R. Ranjan. "GoSharing: An intelligent incentive framework based on users' association for cooperative content sharing in mobile edge networks". In: *Future Generation Comp. Syst.* 95 (2019), pp. 601–614. DOI: 10.1016/j.future.2019.01.013. URL: https://doi.org/10.1016/j.future.2019.01.013 (cit. on p. 153).

[164] A. Mahmood, C. E. Casetti, C.-F. Chiasserini, P. Giaccone, and J. Härri. "The RICH Prefetching in Edge Caches for In-Order Delivery to Connected Cars". In: *IEEE Trans. Vehicular Technology* 68.1 (2019), pp. 4–18. DOI: 10.1109/TVT.2018.2879850. URL: https://doi.org/10.1109/TVT.2018.2879850 (cit. on p. 153).

[165] M. Mamei and F. Zambonelli. "Programming Pervasive and Mobile Computing Applications: The TOTA Approach". In: *ACM Trans. Softw. Eng. Methodol.* 18.4 (July 2009), 15:1–15:56. ISSN: 1049-331X. DOI: 10.1145/1538942.1538945. URL: http://doi.acm.org/10.1145/1538942.1538945 (cit. on pp. 40, 45, 59, 62, 63).

[166] B. Manoj and A. H. Baker. “Communication Challenges in Emergency Response”. In: *Commun. ACM* 50.3 (Mar. 2007), pp. 51–53. ISSN: 0001-0782. DOI: 10.1145/1226736.1226765. URL: http://doi.acm.org/10.1145/1226736.1226765 (cit. on pp. 5, 7, 9, 16, 58).

[167] S. Marreiros. “A Framework for Turn-Based Local Multiplayer Games”. MA thesis. NOVA University Lisbon, Feb. 2021 (cit. on p. 185).

[168] G. B. Mathews. “On the partition of numbers”. In: *Proceedings of the London Mathematical Society* 1.1 (1896), pp. 486–490 (cit. on p. 120).

[169] M. Mauve, A. Widmer, and H. Hartenstein. “A Survey on Position-based Routing in Mobile Ad Hoc Networks”. In: *IEEE Network* 15.6 (Nov. 2001), pp. 30–39. ISSN: 0890-8044. DOI: 10.1109/65.967595. URL: http://dx.doi.org/10.1109/65.967595 (cit. on p. 18).

[170] M. May, G. Karlsson, O. Helgason, and V. Lenders. “A system architecture for delay-tolerant content distribution”. In: *IEEE Conference on Wireless Rural and Emergency Communications*. WreCom ’07. 2007 (cit. on p. 32).

[171] R. Mayer, H. Gupta, E. Saurez, and U. Ramachandran. “FogStore: Toward a Distributed Data Store for Fog Computing”. In: *CoRR* abs/1709.07558 (2017). arXiv: 1709.07558. URL: http://arxiv.org/abs/1709.07558 (cit. on p. 31).

[172] R. Meier and V. Cahill. “STEAM: Event-Based Middleware for Wireless Ad Hoc Network”. In: *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*. ICDCSW ’02. Vienna, Austria: IEEE Computer Society, 2002, pp. 639–644. ISBN: 0-7695-1588-6. URL: http://dl.acm.org/citation.cfm?id=646854.708242 (cit. on p. 28).

[173] G. Metri, A. Agrawal, R. Peri, and W. Shi. “What is eating up battery life on my SmartPhone: A case study”. In: *Proceedings of the International Conference on Energy Aware Computing*. ICEAC ’12. Guzelyurt, Cyprus: IEEE, 2012, pp. 1–6. DOI: 10.1109/ICEAC.2012.6471003. URL: https://doi.org/10.1109/ICEAC.2012.6471003 (cit. on p. 2).

[174] G. Miao, J. Zander, K. W. Sung, and S. B. Slimane. *Fundamentals of Mobile Data Networks*. Cambridge University Press, 2016 (cit. on p. 15).

[175] J. Michel, C. Julien, and J. Payton. “Gander: Mobile, Pervasive Search of the Here and Now in the Here and Now”. In: *IEEE Internet of Things Journal* 1.5 (2014), pp. 483–496. DOI: 10.1109/JIOT.2014.2347132. URL: https://doi.org/10.1109/JIOT.2014.2347132 (cit. on p. 151).

[176] A. Moghadam, S. Srinivasan, and H. Schulzrinne. "7DS - A Modular Platform to Develop Mobile Disruption-Tolerant Applications". In: *Proceedings of the 2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*. NGMAST '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 177–183. ISBN: 978-0-7695-3333-9. DOI: 10.1109/NGMAST.2008.75. URL: https://doi.org/10.1109/NGMAST.2008.75 (cit. on p. 33).

[177] R. Monteiro. "Distributed Storage in a Cloud of Mobile Devices". MA thesis. NOVA University Lisbon, Nov. 2015 (cit. on p. 179).

[178] R. Monteiro, J. A. Silva, J. Lourenço, and H. Paulino. "Decentralized Storage for Networks of Hand-held Devices". In: *Proceedings of the 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. Ed. by P. Zhang, J. S. Silva, N. Lane, F. Boavida, and A. Rodrigues. MobiQuitous '15. Coimbra, Portugal: ICST, 2015, pp. 299–300. DOI: 10.4108/eai.22-7-2015.2260263. URL: https://doi.org/10.4108/eai.22-7-2015.2260263 (cit. on p. 181).

[179] R. Monteiro, J. A. Silva, J. M. Lourenço, and H. Paulino. "Armazenamento Distribuído para Redes de Dispositivos Móveis". In: *Proceedings of the 7th Simpósio Nacional de Informática*. INForum '15. Covilhã, Portugal, 2015 (cit. on p. 180).

[180] A. Montresor and M. Jelasity. "PeerSim: A Scalable P2P Simulator". In: *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*. Seattle, WA, Sept. 2009, pp. 99–100 (cit. on pp. 10, 105, 124, 234).

[181] S. H. Mortazavi, M. Salehe, B. Balasubramanian, E. de Lara, and S. P. Narayanan. "SessionStore: A Session-Aware Datastore for the Edge". In: *4th IEEE International Conference on Fog and Edge Computing*. ICFEC '20. Melbourne, Australia: IEEE, 2020, pp. 59–68. DOI: 10.1109/ICFEC50348.2020.00014. URL: https://doi.org/10.1109/ICFEC50348.2020.00014 (cit. on p. 31).

[182] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips, and E. de Lara. "Cloudpath: a multi-tier cloud computing framework". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. Ed. by J. Zhang, M. Chiang, and B. M. Maggs. SEC '17. ACM, 2017, 20:1–20:13. DOI: 10.1145/3132211.3134464. URL: https://doi.org/10.1145/3132211.3134464 (cit. on p. 31).

[183] G. Mühl. "Large-scale content based publish-subscribe systems". PhD thesis. Darmstadt University of Technology, Germany, 2002. URL: http://elib.tu-darmstadt.de/diss/000274 (cit. on p. 60).

[184] A. Muthitacharoen, S. Gilbert, and R. Morris. *Etna: a Fault-tolerant Algorithm for Atomic Mutable DHT Data*. Tech. rep. MIT CSAIL, 2005 (cit. on p. 38).

[185] D. Neumann, C. Bodenstein, O. F. Rana, and R. Krishnaswamy. "STACEE: Enhancing Storage Clouds Using Edge Devices". In: *Proceedings of the 1st ACM/IEEE Workshop on Autonomic Computing in Economics*. ACE '11. Karlsruhe, Germany: ACM, 2011, pp. 19–26. ISBN: 978-1-4503-0734-5. DOI: 10.1145/1998561.1998567. URL: http://doi.acm.org/10.1145/1998561.1998567 (cit. on p. 31).

[186] E. Nordström, P. Gunningberg, and C. Rohner. "Haggle: A Data-centric Network Architecture for Mobile Devices". In: *Proceedings of the 2009 MobiHoc S3 Workshop on MobiHoc S3*. MobiHoc S3 '09. New Orleans, Louisiana, USA: ACM, 2009, pp. 37–40. ISBN: 978-1-60558-521-5. DOI: 10.1145/1540358.1540370. URL: http://doi.acm.org/10.1145/1540358.1540370 (cit. on p. 32).

[187] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. "The Information Bus: An Architecture for Extensible Distributed Systems". In: *SIGOPS Oper. Syst. Rev.* 27.5 (Dec. 1993), pp. 58–68. ISSN: 0163-5980. DOI: 10.1145/173668.168624. URL: http://doi.acm.org/10.1145/173668.168624 (cit. on pp. 19, 21, 48).

[188] A. Omicini and F. Zambonelli. "Tuple Centres for the Coordination of Internet Agents". In: *Proceedings of the 1999 ACM Symposium on Applied Computing*. SAC '99. San Antonio, Texas, USA: ACM, 1999, pp. 183–190. ISBN: 1-58113-086-4. DOI: 10.1145/298151.298231. URL: http://doi.acm.org/10.1145/298151.298231 (cit. on pp. 40, 45, 62).

[189] OnePlus. *OnePlus 8 Pro*. https://www.oneplus.com/8-pro. Accessed: 2020-09-23. 2020 (cit. on p. 2).

[190] Open Garden, Inc. *FireChat*. https://www.opengarden.com/firechat.html. Accessed: 2018-04-27. 2017 (cit. on p. 63).

[191] J. Ott, E. Hyytia, P. Lassila, T. Vaegs, and J. Kangasharju. "Floating Content: Information Sharing in Urban Areas". In: *Proceedings of the 2011 IEEE International Conference on Pervasive Computing and Communications*. PerCom '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 136–146. ISBN: 978-1-4244-9530-6. DOI: 10.1109/PERCOM.2011.5767578. URL: http://dx.doi.org/10.1109/PERCOM.2011.5767578 (cit. on pp. 33, 151).

[192] J. Paiva, J. Leitão, and L. E. T. Rodrigues. "Rollerchain: A DHT for Efficient Replication". In: *IEEE 12th International Symposium on Network Computing and Applications*. NCA '13. Cambridge, MA, USA: IEEE Computer Society, 2013, pp. 17–24. DOI: 10.1109/NCA.2013.29. URL: https://doi.org/10.1109/NCA.2013.29 (cit. on pp. 38, 72, 105, 108, 118, 124, 126).

[193] J. Paiva and L. E. T. Rodrigues. "Policies for Efficient Data Replication in P2P Systems". In: *19th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2013, Seoul, Korea, December 15-18, 2013*. IEEE Computer Society, 2013, pp. 404–411. DOI: 10.1109/ICPADS.2013.63. URL: https://doi.org/10.1109/ICPADS.2013.63 (cit. on p. 124).

[194] N. P. Palma, V. Mancuso, and M. A. Marsan. “Infrastructureless Pervasive Information Sharing with COTS Devices and Software”. In: *19th IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks*. WoWMoM ’18. Chania, Greece: IEEE Computer Society, 2018, pp. 1–9. DOI: 10.1109/WoWMoM.2018.8449733. URL: https://doi.org/10.1109/WoWMoM.2018.8449733 (cit. on pp. 33, 62).

[195] R. K. Panta, R. Jana, F. Cheng, Y. .-. R. Chen, and V. A. Vaishampayan. “Phoenix: Storage Using an Autonomous Mobile Infrastructure”. In: *IEEE Trans. Parallel Distrib. Syst.* 24.9 (Sept. 2013), pp. 1863–1873. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.84. URL: http://dx.doi.org/10.1109/TPDS.2013.84 (cit. on pp. 30, 61).

[196] K. Paridel, Y. Vanrompay, and Y. Berbers. “Fadip: Lightweight Publish/Subscribe for Mobile Ad Hoc Networks”. In: *Proceedings of the 2010 International Conference on On the Move to Meaningful Internet Systems: Part II*. OTM’10. Hersonissos, Crete, Greece: Springer-Verlag, 2010, pp. 798–810. ISBN: 3-642-16948-1, 978-3-642-16948-9. URL: http://dl.acm.org/citation.cfm?id=1926129.1926142 (cit. on p. 28).

[197] N. W. Paton and O. Díaz. “Active Database Systems”. In: *ACM Comput. Surv.* 31.1 (1999), pp. 63–103. DOI: 10.1145/311531.311623. URL: https://doi.org/10.1145/311531.311623 (cit. on p. 46).

[198] C. Perkins, E. Belding-Royer, and S. Das. *Ad hoc On-Demand Distance Vector (AODV) Routing*. RFC 3561. http://www.rfc-editor.org/rfc/rfc3561.txt. RFC Editor, July 2003. URL: http://www.rfc-editor.org/rfc/rfc3561.txt (cit. on p. 18).

[199] C. E. Perkins. *Ad Hoc Networking*. 1st ed. Addison-Wesley Professional, 2008. ISBN: 0321579070, 9780321579072 (cit. on p. 13).

[200] C. E. Perkins and P. Bhagwat. “Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers”. In: *Proceedings of the Conference on Communications Architectures, Protocols and Applications*. SIGCOMM ’94. London, United Kingdom: ACM, 1994, pp. 234–244. ISBN: 0-89791-682-4. DOI: 10.1145/190314.190336. URL: http://doi.acm.org/10.1145/190314.190336 (cit. on pp. 17, 65, 84).

[201] Pew Research Center. *Social Media Update 2016*. https://goo.gl/eeFs0c. Accessed: 2020-03-28. 2016 (cit. on p. 2).

[202] G. P. Picco, A. L. Murphy, and G.-C. Roman. “LIME: Linda Meets Mobility”. In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE ’99. Los Angeles, California, USA: ACM, 1999, pp. 368–377. ISBN: 1-58113-074-0. DOI: 10.1145/302405.302659. URL: http://doi.acm.org/10.1145/302405.302659 (cit. on pp. 40, 45, 59, 62).

[203] P. R. Pietzuch and J. Bacon. "Hermes: A Distributed Event-Based Middleware Architecture". In: *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*. ICDCSW '02. Vienna, Austria: IEEE Computer Society, 2002, pp. 611–618. ISBN: 0-7695-1588-6. DOI: 10.1109/ICDCSW.2002.1030837. URL: http://dl.acm.org/citation.cfm?id=646854.708058 (cit. on pp. 22, 24, 27, 70).

[204] D. Powell. "Group Communication". In: *Commun. ACM* 39.4 (Apr. 1996), pp. 50–53. ISSN: 0001-0782. DOI: 10.1145/227210.227225. URL: http://doi.acm.org/10.1145/227210.227225 (cit. on p. 21).

[205] M. A. Qader and V. Hristidis. "DualDB: An Efficient LSM-based Publish/Subscribe Storage System". In: *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. SSDBM '17. Chicago, IL, USA: ACM, 2017, pp. 1–6. ISBN: 978-1-4503-5282-6. DOI: 10.1145/3085504.3085528. URL: http://doi.acm.org/10.1145/3085504.3085528 (cit. on p. 28).

[206] A. Rai, K. K. Chintalapudi, V. N. Padmanabhan, and R. Sen. "Zee: Zero-effort Crowdsourcing for Indoor Localization". In: *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*. Mobicom '12. Istanbul, Turkey: ACM, 2012, pp. 293–304. ISBN: 978-1-4503-1159-5. DOI: 10.1145/2348543.2348580. URL: http://doi.acm.org/10.1145/2348543.2348580 (cit. on pp. 18, 65, 76).

[207] A. Raman, N. Sastry, A. Sathiaseelan, J. Chandaria, and A. Secker. "Wi-Stitch: Content Delivery in Converged Edge Networks". In: *Proceedings of the Workshop on Mobile Edge Communications*. MECOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 13–18. ISBN: 978-1-4503-5052-5. DOI: 10.1145/3098208.3098211. URL: http://doi.acm.org/10.1145/3098208.3098211 (cit. on p. 31).

[208] V. Ramasubramanian and E. G. Sirer. "Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays". In: *1st Symposium on Networked Systems Design and Implementation*. NSDI '04. San Francisco, California, USA, 2004, pp. 99–112. URL: http://www.usenix.org/events/nsdi04/tech/ramasubramanian.html (cit. on p. 37).

[209] L. Ramaswamy and J. Chen. "The CoQUOS Approach to Continuous Queries in Unstructured Overlays". In: *IEEE Trans. on Knowl. and Data Eng.* 23.3 (Mar. 2011), pp. 463–478. ISSN: 1041-4347. DOI: 10.1109/TKDE.2010.133. URL: http://dx.doi.org/10.1109/TKDE.2010.133 (cit. on p. 32).

[210] A. Rao, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica. "Load Balancing in Structured P2P Systems". In: *Peer-to-Peer Systems II, Second International Workshop*. IPTPS '03. Berkeley, California, USA, 2003, pp. 68–79. DOI: 10.1007/978-3-540-45172-3_6. URL: https://doi.org/10.1007/978-3-540-45172-3%5C_6 (cit. on p. 37).

[211] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. “A scalable content-addressable network”. In: *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Ed. by R. L. Cruz and G. Varghese. SIGCOMM ’01. San Diego, California, USA: ACM, 2001, pp. 161–172. DOI: 10.1145/383059.383072. URL: https://doi.org/10.1145/383059.383072 (cit. on pp. 34, 104).

[212] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. “GHT: A Geographic Hash Table for Data-centric Storage”. In: *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*. WSNA ’02. Atlanta, Georgia, USA: ACM, 2002, pp. 78–87. ISBN: 1-58113-589-0. DOI: 10.1145/570738.570750. URL: http://doi.acm.org/10.1145/570738.570750 (cit. on pp. 59, 65, 72, 73, 79).

[213] D. Raychaudhuri, K. Nagaraja, and A. Venkataramani. “MobilityFirst: A Robust and Trustworthy Mobility-centric Architecture for the Future Internet”. In: *SIGMOBILE Mob. Comput. Commun. Rev.* 16.3 (Dec. 2012), pp. 2–13. ISSN: 1559-1662. DOI: 10.1145/2412096.2412098. URL: http://doi.acm.org/10.1145/2412096.2412098 (cit. on p. 152).

[214] D. Remédios, A. Teófilo, H. Paulino, and J. Lourenço. “Mobile Device-to-Device Distributed Computing Using Data Sets”. In: *Proceedings of the 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. Ed. by P. Zhang, J. S. Silva, N. Lane, F. Boavida, and A. Rodrigues. MobiQuitous ’15. Coimbra, Portugal: ICST, 2015, pp. 297–298. DOI: 10.4108/eai.22-7-2015.2260273. URL: https://doi.org/10.4108/eai.22-7-2015.2260273 (cit. on p. 183).

[215] P. Reynolds and A. Vahdat. “Efficient Peer-to-Peer Keyword Searching”. In: *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*. Ed. by M. Endler and D. C. Schmidt. Vol. 2672. Lecture Notes in Computer Science. Rio de Janeiro, Brazil: Springer, 2003, pp. 21–40. DOI: 10.1007/3-540-44892-6_2. URL: https://doi.org/10.1007/3-540-44892-6%5C_2 (cit. on pp. 36, 104, 105).

[216] G. F. Riley and T. R. Henderson. “The ns-3 Network Simulator”. In: *Modeling and Tools for Network Simulation*. Ed. by K. Wehrle, M. Güneş, and J. Gross. Springer Berlin Heidelberg, 2010, pp. 15–34. ISBN: 978-3-642-12331-3. DOI: 10.1007/978-3-642-12331-3_2. URL: http://dx.doi.org/10.1007/978-3-642-12331-3_2 (cit. on pp. 9, 59, 75, 83).

[217] J. Rodrigues, E. R. B. Marques, L. M. B. Lopes, and F. Silva. “Towards a Middleware for Mobile Edge-cloud Applications”. In: *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets*. MECC ’17. Las Vegas, Nevada, USA:

ACM, 2017, 1:1–1:6. ISBN: 978-1-4503-5171-3. DOI: 10.1145/3152360.3152361. URL: http://doi.acm.org/10.1145/3152360.3152361 (cit. on pp. 64, 77).

[218] J. Rodrigues, E. R. B. Marques, J. Silva, L. M. B. Lopes, and F. M. A. Silva. "Video Dissemination in Untethered Edge-Clouds: A Case Study". In: *International Conference on Distributed Applications and Interoperable Systems*. Ed. by S. Bonomi and E. Rivière. Vol. 10853. DAIS '18. Madrid, Spain: Springer, 2018, pp. 137–152. DOI: 10.1007/978-3-319-93767-0_10. URL: https://doi.org/10.1007/978-3-319-93767-0%5C_10 (cit. on pp. 125, 153).

[219] J. Rodrigues, J. Silva, R. Martins, L. M. B. Lopes, U. Drolia, P. Narasimhan, and F. M. A. Silva. "Benchmarking Wireless Protocols for Feasibility in Supporting Crowdsourced Mobile Computing". In: *International Conference on Distributed Applications and Interoperable Systems*. Ed. by M. Jelasity and E. Kalyvianaki. Vol. 9687. DAIS '16. Heraklion, Crete, Greece: Springer, 2016, pp. 96–108. ISBN: 978-3-319-39576-0. DOI: 10.1007/978-3-319-39577-7_8. URL: https://doi.org/10.1007/978-3-319-39577-7%5C_8 (cit. on p. 195).

[220] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. "Stronger Password Authentication Using Browser Extensions". In: *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. Ed. by P. D. McDaniel. USENIX Association, 2005. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/stronger-password-authentication-using-browser-extensions (cit. on p. 120).

[221] A. I. T. Rowstron and P. Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms*. Ed. by R. Guerraoui. Vol. 2218. Lecture Notes in Computer Science. Heidelberg, Germany: Springer, 2001, pp. 329–350. DOI: 10.1007/3-540-45518-3_18. URL: https://doi.org/10.1007/3-540-45518-3%5C_18 (cit. on pp. 34, 104).

[222] J. H. Saltzer, D. P. Reed, and D. D. Clark. "End-to-end Arguments in System Design". In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984), pp. 277–288. ISSN: 0734-2071. DOI: 10.1145/357401.357402. URL: http://doi.acm.org/10.1145/357401.357402 (cit. on p. 66).

[223] P. Sanches. "Distributed Computing in a Cloud of Mobile Phones". http://hdl.handle.net/10362/30063. MA thesis. NOVA University Lisbon, Sept. 2017 (cit. on p. 182).

[224] P. Sanches, J. A. Silva, A. Teófilo, and H. Paulino. "Data-Centric Distributed Computing on Networks of Mobile Devices". In: *26th International Conference on Parallel and Distributed Computing*. Ed. by M. Malawski and K. Rzadca. Vol. 12247. Euro-Par '20. Warsaw, Poland (Online): Springer, 2020, pp. 296–311. DOI: 10.10

07/978-3-030-57675-2_19. URL: https://doi.org/10.1007/978-3-030-57675-2%5C_19 (cit. on p. 183).

[225] P. Sanches, A. Teófilo, F. Cerqueira, J. A. Silva, and H. Paulino. "Computação Distribuída em Redes Formadas por Dispositivos Móveis". In: *Proceedings of the 9th Simpósio Nacional de Informática*. INForum '17. Aveiro, Portugal, 2017 (cit. on p. 183).

[226] A. A. Sani, W. Richter, X. Bao, T. Narayan, M. Satyanarayanan, L. Zhong, and R. R. Choudhury. "Opportunistic Content Search of Smartphone Photos". In: *CoRR* abs/1106.5568 (2011). arXiv: 1106.5568. URL: http://arxiv.org/abs/1106.5568 (cit. on p. 32).

[227] S. K. Sarkar, T. G. Basavaraju, and C. Puttamadappa. *Ad hoc mobile wireless networks: principles, protocols and applications*. CRC Press, 2007. ISBN: 1420062220, 9781420062229 (cit. on pp. 14–16).

[228] L. F. G. Sarmenta. "Bayanihan: Web-Based Volunteer Computing Using Java". In: *Proceedings of the Second International Conference on Worldwide Computing and Its Applications*. Ed. by Y. Masunaga, T. Katayama, and M. Tsukamoto. Vol. 1368. WWCA '98. Tsukuba, Japan: Springer-Verlag, 1998, pp. 444–461. ISBN: 3-540-64216-1. DOI: 10.1007/3-540-64216-1_67. URL: https://doi.org/10.1007/3-540-64216-1%5C_67 (cit. on p. 54).

[229] S. Saroiu, P. K. Gummadi, and S. D. Gribble. "Measuring and analyzing the characteristics of Napster and Gnutella hosts". In: *Multim. Syst.* 9.2 (2003), pp. 170–184. DOI: 10.1007/s00530-003-0088-1. URL: https://doi.org/10.1007/s00530-003-0088-1 (cit. on pp. 33, 34).

[230] M. Satyanarayanan. "Fundamental Challenges in Mobile Computing". In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 1–7. ISBN: 0-89791-800-2. DOI: 10.1145/248052.248053. URL: http://doi.acm.org/10.1145/248052.248053 (cit. on p. 1).

[231] M. Satyanarayanan. "Cloudlets: At the Leading Edge of Cloud-Mobile Convergence". In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. QoSA '13. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2013, pp. 1–2. ISBN: 9781450321266. DOI: 10.1145/2465478.2465494. URL: https://doi.org/10.1145/2465478.2465494 (cit. on p. 3).

[232] M. Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (Jan. 2017), pp. 30–39. ISSN: 0018-9162. DOI: 10.1109/MC.2017.9. URL: https://doi.org/10.1109/MC.2017.9 (cit. on p. 3).

[233] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. “The Case for VM-Based Cloudlets in Mobile Computing”. In: *IEEE Pervasive Computing* 8.4 (Oct. 2009), pp. 14–23. ISSN: 1536-1268. DOI: 10.1109/MPRV.2009.82. URL: http://dx.doi.org/10.1109/MPRV.2009.82 (cit. on p. 3).

[234] R. Schollmeier. “A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications”. In: *1st International Conference on Peer-to-Peer Computing*. Ed. by R. L. Graham and N. Shahmehri. P2P ’01. Linköping, Sweden: IEEE Computer Society, 2001, pp. 101–102. DOI: 10.1109/P2P.2001.990434. URL: https://doi.org/10.1109/P2P.2001.990434 (cit. on p. 33).

[235] T. Schütt, F. Schintke, and A. Reinefeld. “Structured Overlay without Consistent Hashing: Empirical Results”. In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), 16-19 May 2006, Singapore*. IEEE Computer Society, 2006, p. 8. DOI: 10.1109/CCGRID.2006.175. URL: http://doi.ieeecomputersociety.org/10.1109/CCGRID.2006.175 (cit. on p. 118).

[236] K. Seada and A. Helmy. “Rendezvous regions: a scalable architecture for service location and data-centric storage in large-scale wireless networks”. In: *18th International Parallel and Distributed Processing Symposium*. IPDPS ’04. Santa Fe, New Mexico, USA: IEEE Computer Society, Apr. 2004, pp. 218–. DOI: 10.1109/IPDPS.2004.1303252. URL: https://doi.org/10.1109/IPDPS.2004.1303252 (cit. on p. 72).

[237] K. Seada and C. Perkins. *Social networks: the killer app for wireless ad hoc networks?* Tech. rep. Nokia Research Centre, 2006 (cit. on p. 64).

[238] B. Segall and D. Arnold. “Elvin has left the building: A publish/subscribe notification service with quenching”. In: AUUG ’97 (1997), pp. 243–255 (cit. on pp. 22, 25).

[239] V. Setty, G. Kreitz, R. Vitenberg, M. van Steen, G. Urdaneta, and S. Gimåker. “The Hidden Pub/Sub of Spotify: (Industry Article)”. In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS ’13. Arlington, Texas, USA: ACM, 2013, pp. 231–240. ISBN: 978-1-4503-1758-0. DOI: 10.1145/2488222.2488273. URL: http://doi.acm.org/10.1145/2488222.2488273 (cit. on p. 29).

[240] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. “Edge Computing: Vision and Challenges”. In: *IEEE Internet Things J.* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198. URL: https://doi.org/10.1109/JIOT.2016.2579198 (cit. on pp. 3, 58, 104).

[241] J. A. Silva, F. Cerqueira, H. Paulino, J. M. Lourenço, J. Leitão, and N. Preguiça. "It's about Thyme: On the design and implementation of a time-aware reactive storage system for pervasive edge computing environments". In: *Future Generation Computer Systems* 118 (May 2021), pp. 14–36. ISSN: 0167-739X. DOI: 10.1016/j.future.2020.12.008. URL: http://www.sciencedirect.com/science/article/pii/S0167739X20330703 (cit. on pp. 54, 100).

[242] J. A. Silva, J. Leitão, N. Preguiça, J. M. Lourenço, and H. Paulino. "Towards the Opportunistic Combination of Mobile Ad-hoc Networks with Infrastructure Access". In: *Proceedings of the 1st Workshop on Middleware for Edge Clouds & Cloudlets*. MECC '16. Trento, Italy: ACM, 2016, pp. 1–3. ISBN: 978-1-4503-4668-9. DOI: 10.1145/3017116.3022873. URL: http://doi.acm.org/10.1145/3017116.3022873 (cit. on pp. 64, 100, 182).

[243] J. A. Silva, R. Monteiro, H. Paulino, and J. M. Lourenço. "Ephemeral Data Storage for Networks of Hand-Held Devices". In: *IEEE Trustcom/BigDataSE/ISPA*. Tianjin, China: IEEE, 2016, pp. 1106–1113. DOI: 10.1109/TrustCom.2016.0182. URL: https://doi.org/10.1109/TrustCom.2016.0182 (cit. on pp. 30, 57, 61, 150, 180).

[244] J. A. Silva, H. Paulino, J. M. Lourenço, J. Leitão, and N. M. Preguiça. "Time-Aware Publish/Subscribe for Networks of Mobile Devices". In: *CoRR* abs/1801.00297 (Dec. 2017). arXiv: 1801.00297. URL: http://arxiv.org/abs/1801.00297 (cit. on pp. 55, 101).

[245] J. A. Silva, H. Paulino, J. M. Lourenço, J. Leitão, and N. M. Preguiça. "Time-aware reactive storage in wireless edge environments". In: *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. Ed. by H. V. Poor, Z. Han, D. Pompili, Z. Sun, and M. Pan. MobiQuitous '19. Houston, Texas, USA: ACM, 2019, pp. 238–247. DOI: 10.1145/3360774.3360828. URL: https://doi.org/10.1145/3360774.3360828 (cit. on pp. 54, 100, 151).

[246] J. A. Silva, P. Vieira, and H. Paulino. "Data Storage and Sharing for Mobile Devices in Multi-region Edge Networks". In: *21st IEEE International Symposium on "A World of Wireless, Mobile and Multimedia Networks"*. WoWMoM '20. Cork, Ireland (Online): IEEE, 2020, pp. 40–49. DOI: 10.1109/WoWMoM49955.2020.00021. URL: https://doi.org/10.1109/WoWMoM49955.2020.00021 (cit. on p. 177).

[247] P. M. P. Silva, J. Rodrigues, J. Silva, R. Martins, L. M. B. Lopes, and F. M. A. Silva. "Using Edge-Clouds to Reduce Load on Traditional WiFi Infrastructures and Improve Quality of Experience". In: *1st IEEE International Conference on Fog and Edge Computing*. ICFEC '17. Madrid, Spain: IEEE Computer Society, 2017, pp. 61–67. DOI: 10.1109/ICFEC.2017.14. URL: https://doi.org/10.1109/ICFEC.2017.14 (cit. on p. 153).

[248] J. Singh, D. M. Eyers, and J. Bacon. “Controlling Historical Information Dissemination in Publish/Subscribe”. In: *Proceedings of the 2008 Workshop on Middleware Security*. MidSec ’08. Leuven, Belgium: ACM, 2008, pp. 34–39. ISBN: 978-1-60558-363-1. DOI: 10.1145/1463342.1463349. URL: http://doi.acm.org/10.1145/1463342.1463349 (cit. on p. 29).

[249] X. Song, Y. Huang, Q. Zhou, F. Ye, Y. Yang, and X. Li. “Content Centric Peer Data Sharing in Pervasive Edge Computing Environments”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems*. ICDCS ’17. Atlanta, GA, USA: IEEE Computer Society, June 2017, pp. 287–297. DOI: 10.1109/ICDCS.2017.26. URL: https://doi.org/10.1109/ICDCS.2017.26 (cit. on pp. 32, 43, 59, 61, 150–152).

[250] X. Song, Y. Huang, Q. Zhou, F. Ye, Y. Yang, and X. Li. “Pervasive edge data sharing in MANET”. In: *2017 IEEE Conference on Computer Communications Workshops*. Atlanta, GA, USA: IEEE, May 2017, pp. 133–138. DOI: 10.1109/INFCOMW.2017.8116365. URL: https://doi.org/10.1109/INFCOMW.2017.8116365 (cit. on p. 32).

[251] V. Sourlas, P. Flegkas, G. S. Paschos, D. Katsaros, and L. Tassiulas. “Storing and Replication in Topic-Based Publish/Subscribe Networks”. In: *Proceedings of the 29th IEEE Conference on Global Telecommunications*. GLOBECOM’10. Miami, Florida, USA: IEEE, Dec. 2010, pp. 1–5. DOI: 10.1109/GLOCOM.2010.5683977. URL: https://doi.org/10.1109/GLOCOM.2010.5683977 (cit. on p. 28).

[252] V. Sourlas, G. S. Paschos, P. Flegkas, and L. Tassiulas. “Caching in Content-based Publish/Subscribe Systems”. In: *Proceedings of the 28th IEEE Conference on Global Telecommunications*. GLOBECOM’09. Honolulu, Hawaii, USA: IEEE Press, 2009, pp. 1401–1406. ISBN: 978-1-4244-4147-1. URL: http://dl.acm.org/citation.cfm?id=1811380.1811612 (cit. on p. 28).

[253] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner. “Mires: A Publish/Subscribe Middleware for Sensor Networks”. In: *Personal Ubiquitous Comput.* 10.1 (Dec. 2005), pp. 37–44. ISSN: 1617-4909. DOI: 10.1007/s00779-005-0038-3. URL: http://dx.doi.org/10.1007/s00779-005-0038-3 (cit. on p. 28).

[254] K. Sripanidkulchai, B. M. Maggs, and H. Zhang. “Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems”. In: *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*. INFOCOM ’03. San Francisco, California, USA: IEEE Computer Society, 2003, pp. 2166–2176. DOI: 10.1109/INFCOM.2003.1209237. URL: https://doi.org/10.1109/INFCOM.2003.1209237 (cit. on p. 7).

[255] T. Stading, P. Maniatis, and M. Baker. “Peer-to-Peer Caching Schemes to Address Flash Crowds”. In: *Peer-to-Peer Systems, First International Workshop*. IPTPS ’02. Cambridge, MA, USA, 2002, pp. 203–213. DOI: 10.1007/3-540-45748-8_19. URL: https://doi.org/10.1007/3-540-45748-8%5C_19 (cit. on p. 37).

[256] I. Stoica, R. T. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. “Chord: A scalable peer-to-peer lookup service for internet applications”. In: *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Ed. by R. L. Cruz and G. Varghese. SIGCOMM ’01. San Diego, California, USA: ACM, 2001, pp. 149–160. DOI: 10.1145/383059.383071. URL: https://doi.org/10.1145/383059.383071 (cit. on pp. xxvii, 34–36, 104, 108, 118).

[257] I. Stojmenovic. “Position-based Routing in Ad Hoc Networks”. In: *Comm. Mag.* 40.7 (July 2002), pp. 128–134. ISSN: 0163-6804. DOI: 10.1109/MCOM.2002.1018018. URL: http://dx.doi.org/10.1109/MCOM.2002.1018018 (cit. on pp. 18, 41).

[258] D. Stutzbach and R. Rejaie. “Understanding churn in peer-to-peer networks”. In: *Proceedings of the 6th ACM SIGCOMM Internet Measurement Conference*. Ed. by J. M. Almeida, V. A. F. Almeida, and P. Barford. IMC ’06. Rio de Janeriro, Brazil: ACM, 2006, pp. 189–202. DOI: 10.1145/1177080.1177105. URL: https://doi.org/10.1145/1177080.1177105 (cit. on p. 33).

[259] J. Su, J. Scott, P. Hui, J. Crowcroft, E. De Lara, C. Diot, A. Goel, M. H. Lim, and E. Upton. “Haggle: Seamless Networking for Mobile Applications”. In: *Proceedings of the 9th International Conference on Ubiquitous Computing*. UbiComp ’07. Innsbruck, Austria: Springer-Verlag, 2007, pp. 391–408. ISBN: 978-3-540-74852-6. URL: http://dl.acm.org/citation.cfm?id=1771592.1771615 (cit. on p. 62).

[260] S. Al-Sultan, M. M. Al-Doori, A. H. Al-Bayatti, and H. Zedan. “A comprehensive survey on vehicular Ad Hoc network”. In: *Journal of Network and Computer Applications* 37 (2014), pp. 380–392. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2013.02.036. URL: https://doi.org/10.1016/j.jnca.2013.02.036 (cit. on p. 16).

[261] P. Sutton, R. Arkins, and B. Segall. “Supporting Disconnectedness-Transparent Information Delivery for Mobile and Invisible Computing”. In: *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*. CCGRID ’01. Brisbane, Australia: IEEE, 2001, pp. 277–287. ISBN: 0-7695-1010-8. DOI: 10.1109/CCGRID.2001.923204. URL: http://dl.acm.org/citation.cfm?id=560889.792416 (cit. on p. 28).

[262] T. Taleb and A. Ksentini. “Follow me cloud: interworking federated clouds and distributed mobile networks”. In: *IEEE Network* 27.5 (Sept. 2013), pp. 12–19. ISSN: 0890-8044. DOI: 10.1109/MNET.2013.6616110. URL: https://doi.org/10.1109/MNET.2013.6616110 (cit. on p. 3).

[263] C. Tang and S. Dwarkadas. "Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval". In: *1st Symposium on Networked Systems Design and Implementation*. Ed. by R. T. Morris and S. Savage. NSDI '04. San Francisco, California, USA: USENIX, 2004, pp. 211–224. URL: http://www.usenix.org/events/nsdi04/tech/tang.html (cit. on pp. 36, 104, 105).

[264] A. Teófilo, H. Paulino, and J. M. Lourenço. "RedMesh: A WiFi-Direct Network Formation Algorithm for Large-Scale Scenarios". In: *Proceedings of the 17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. MobiQuitous '20. ACM, 2020 (cit. on p. 64).

[265] A. Teófilo, D. Remédios, J. M. Lourenço, and H. Paulino. "GOCRGO and GOGO: Two Minimal Communication Topologies for WiFi-Direct Multi-group Networking". In: *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. Ed. by T. Gu, R. Kotagiri, and H. Liu. MobiQuitous '17. Melbourne, Australia: ACM, 2017, pp. 232–241. DOI: 10.1145/3144457.3144481. URL: https://doi.org/10.1145/3144457.3144481 (cit. on pp. 64, 155, 195).

[266] D. B. Terry, D. Goldberg, D. A. Nichols, and B. M. Oki. "Continuous Queries over Append-Only Databases". In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. Ed. by M. Stonebraker. SIGMOD '92. San Diego, California, USA: ACM Press, 1992, pp. 321–330. DOI: 10.1145/130283.130333. URL: https://doi.org/10.1145/130283.130333 (cit. on p. 47).

[267] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. "Consistency-based service level agreements for cloud storage". In: *ACM SIGOPS 24th Symposium on Operating Systems Principles*. Ed. by M. Kaminsky and M. Dahlin. SOSP '13. Farmington, PA, USA: ACM, 2013, pp. 309–324. DOI: 10.1145/2517349.2522731. URL: https://doi.org/10.1145/2517349.2522731 (cit. on p. 195).

[268] K. Thilakarathna, H. Petander, J. Mestre, and A. Seneviratne. "MobiTribe: Cost Efficient Distributed User Generated Content Sharing on Smartphones". In: *IEEE Transactions on Mobile Computing* 13.9 (Sept. 2014), pp. 2058–2070. ISSN: 1536-1233. DOI: 10.1109/TMC.2013.89. URL: https://doi.org/10.1109/TMC.2013.89 (cit. on pp. 30, 61).

[269] K. Thilakarathna, F.-Z. Jiang, S. Mrabet, M. Ali Kaafar, A. Seneviratne, and G. Xie. "Crowd-Cache: Leveraging on spatio-temporal correlation in content popularity for mobile networking in proximity". In: *Computer Communications* 100.C (Mar. 2017), pp. 104–117. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2017.01.006. URL: https://doi.org/10.1016/j.comcom.2017.01.006 (cit. on p. 31).

[270] Tibco Software Inc. *Tibco Rendezvous*. https://www.tibco.com/products/tibco-rendezvous. Accessed: 2020-01-30. 2018 (cit. on pp. 21, 25).

[271] L. Tornqvist, P. Vartia, and Y. O. Vartia. “How Should Relative Changes Be Measured?” In: *The American Statistician* 39.1 (1985), pp. 43–46. ISSN: 00031305. URL: http://www.jstor.org/stable/2683905 (cit. on p. 128).

[272] Tox. *Tox instant messaging*. https://tox.chat/. Accessed: 2019-06-23. 2019 (cit. on p. 104).

[273] 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 (cit. on pp. 151, 152).

[274] A. Trivedi, L. Wang, H. E. Bal, and A. Iosup. “Sharing and Caring of Data at the Edge”. In: *3rd USENIX Workshop on Hot Topics in Edge Computing*. Ed. by I. Ahmad and M. Zhao. HotEdge ’20. USENIX Association, 2020. URL: https://www.usenix.org/conference/hotedge20/presentation/trivedi (cit. on p. 31).

[275] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, and A. C. Rice. “Exhausting battery statistics: understanding the energy demands on mobile handsets”. In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Networking, Systems, and Applications for Mobile Handhelds*. Ed. by L. P. Cox and A. Wolman. MobiHeld ’10. New Delhi, India: ACM, 2010, pp. 9–14. DOI: 10.1145/1851322.1851327. URL: https://doi.org/10.1145/1851322.1851327 (cit. on p. 2).

[276] L. M. Vaquero and L. Rodero-Merino. “Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing”. In: *SIGCOMM Comput. Commun. Rev.* 44.5 (Oct. 2014), pp. 27–32. ISSN: 0146-4833. DOI: 10.1145/2677046.2677052. URL: http://doi.acm.org/10.1145/2677046.2677052 (cit. on p. 3).

[277] L. Vargas, J. Bacon, and K. Moody. “Integrating Databases with Publish/Subscribe”. In: *Proceedings of the Fourth International Workshop on Distributed Event-Based Systems*. ICDCSW ’05. Columbus, Ohio, USA: IEEE Computer Society, 2005, pp. 392–397. ISBN: 0-7695-2328-5-04. DOI: 10.1109/ICDCSW.2005.79. URL: https://doi.org/10.1109/ICDCSW.2005.79 (cit. on pp. 29, 44, 45, 60).

[278] P. Vieira. “A Persistent Publish/Subscribe System for Mobile Edge Computing”. http://hdl.handle.net/10362/71124. MA thesis. NOVA University Lisbon, Dec. 2018 (cit. on pp. 10, 151, 155, 159, 163, 165, 167–176).

[279] 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 (cit. on pp. 151, 152).

[280] S. Wang, K. S. Chan, R. Urgaonkar, T. He, and K. K. Leung. “Emulation-based study of dynamic service placement in mobile micro-clouds”. In: *34th IEEE Military Communications Conference*. Ed. by Q. Zhang, J. Brand, T. G. MacDonald, B. T. Doshi, and B. L. Gorsic. MILCOM ’15. Tampa, Florida, USA: IEEE, Oct. 2015, pp. 1046–1051. DOI: 10.1109/MILCOM.2015.7357583. URL: https://doi.org/10.1109/MILCOM.2015.7357583 (cit. on p. 3).

[281] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang. “A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications”. In: *IEEE Access* 5 (2017), pp. 6757–6779. DOI: 10.1109/ACCESS.2017.2685434. URL: https://doi.org/10.1109/ACCESS.2017.2685434 (cit. on p. 153).

[282] 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: *59th IEEE Global Communications Conference*. GLOBECOM ’16. 2016, pp. 1–7. DOI: 10.1109/GLOCOM.2016.7842342. URL: https://doi.org/10.1109/GLOCOM.2016.7842342 (cit. on pp. 150–152).

[283] 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 (cit. on p. 151).

[284] 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: *1st IEEE International Conference on Edge Computing*. EDGE ’17. IEEE, 2017, pp. 40–46. DOI: 10.1109/IEEE.EDGE.2017.14. URL: https://doi.org/10.1109/IEEE.EDGE.2017.14 (cit. on pp. 151, 152).

[285] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos. “A Survey of Information-Centric Networking Research”. In: *IEEE Commun. Surv. Tutorials* 16.2 (2014), pp. 1024–1049. DOI: 10.1109/SURV.2013.070813.00063. URL: https://doi.org/10.1109/SURV.2013.070813.00063 (cit. on p. 61).

[286] Y. Yan, N. H. Tran, and F. S. Bao. “Gossiping along the Path: A Direction-Biased Routing Scheme for Wireless Ad Hoc Networks”. In: *2015 IEEE Global Communications Conference*. GLOBECOM ’15. San Diego, California, USA: IEEE, 2015, pp. 1–6. DOI: 10.1109/GLOCOM.2014.7417867. URL: https://doi.org/10.1109/GLOCOM.2014.7417867 (cit. on p. 58).

[287] S. Yi, C. Li, and Q. Li. “A Survey of Fog Computing: Concepts, Applications and Issues”. In: *Proceedings of the 2015 Workshop on Mobile Big Data*. Mobidata ’15. Hangzhou, China: ACM, 2015, pp. 37–42. ISBN: 978-1-4503-3524-9. DOI: 10.1145/2757384.2757397. URL: http://doi.acm.org/10.1145/2757384.2757397 (cit. on p. 3).

[288] J. Yick, B. Mukherjee, and D. Ghosal. “Wireless sensor network survey”. In: *Comput. Networks* 52.12 (2008), pp. 2292–2330. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2008.04.002. URL: https://doi.org/10.1016/j.comnet.2008.04.002 (cit. on p. 16).

[289] Yinzcam, Inc. *Yinzcam*. http://www.yinzcam.com/. Accessed: 2020-08-13. Nov. 2019 (cit. on p. 64).

[290] E. Yoneki, P. Hui, S. Y. Chan, and J. Crowcroft. “A socio-aware overlay for publish/subscribe communication in delay tolerant networks”. In: *10th International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. MSWiM ’07. 2007, pp. 225–234. DOI: 10.1145/1298126.1298166. URL: https://doi.org/10.1145/1298126.1298166 (cit. on p. 151).

[291] T. Zahn and J. Schiller. “MADPastry: A DHT Substrate for Practicably Sized MANETs”. In: *Proceedings of the 5th Workshop on Applications and Services in Wireless Networks*. ASWN ’05. 2005 (cit. on p. 72).

[292] 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 ’15. Boston, MA, USA: IEEE Computer Society, 2015, pp. 1–9. DOI: 10.1109/WoWMoM.2015.7158137. URL: https://doi.org/10.1109/WoWMoM.2015.7158137 (cit. on pp. 31, 150–152).

[293] K. Zhang, S. Leng, Y. He, S. Maharjan, and Y. Zhang. “Cooperative Content Caching in 5G Networks with Mobile Edge Computing”. In: *IEEE Wireless Commun.* 25.3 (2018), pp. 80–87. DOI: 10.1109/MWC.2018.1700303. URL: https://doi.org/10.1109/MWC.2018.1700303 (cit. on p. 153).

[294] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. C. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang. “Named data networking”. In: *Comput. Commun. Rev.* 44.3 (2014), pp. 66–73. DOI: 10.1145/2656877.2656887. URL: https://doi.org/10.1145/2656877.2656887 (cit. on p. 39).

[295] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. “Tapestry: a resilient global-scale overlay for service deployment”. In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 41–53. DOI: 10.1109/JSAC.2003.818784. URL: https://doi.org/10.1109/JSAC.2003.818784 (cit. on p. 34).

[296] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. “Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination”. In: *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV ’01. Port Jefferson, New York, USA:

ACM, 2001, pp. 11–20. ISBN: 1-58113-370-7. DOI: 10.1145/378344.378347. URL: http://doi.acm.org/10.1145/378344.378347 (cit. on pp. 21, 27).

[297] G. K. Zipf. *Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology*. Addison-Wesley, 1949 (cit. on p. 104).

APPENDIX A

PARSLEY'S GROUP SIZE STUDY

"Man's real home is not a house, but the road."
— Bruce Chatwin

Presented in §5, PARSLEY is our proposal on a resilient group-based distributed hash table (DHT) with a preemptive peer relocation (PPR) technique and a dynamic data sharding mechanism. Besides the minimum and maximum group size hard limits (l and h, respectively) entailed by the group-based approach, the PPR feature encompasses other two soft limits (l' and h'). These soft limits define a desired target interval for group size, allowing the overlay to take some preemptive measures before reaching the hard limits. Thus, we end up with four parameters that need to be defined. Since it is unfeasible to evaluate all the possible values for each parameter, in this appendix, we shed some light on the reasons behind the values used in PARSLEY's evaluation reported in §5.5.

In their respective evaluations, related works (discussed in §5.2) define the group size limits they used (e.g., Rollerchain uses 3–8, and MobiStore uses 2–25). However, they never justify the chosen values. On the contrary, here, we do an overlay characterization study regarding the group size parameters and lay our rationale.

We present the setup used for these experiments in §A.1. Next, §A.2 reports the results regarding topology operations, and §A.3 addresses the highs and lows concerning big groups. After, in §A.4, we do a broad discussion about our major findings. Lastly, we present some of the complete plots that did not fit in the previous sections in §A.5.

A.1 Experimental Setup

For this study, we use the same experimental setup described in §5.5.1, and the same experiment scenarios as detailed in §5.5.2. We use a system comprised by 10 000 peers, populated with 50 000 values distributed among 10 000 keys, and values are assigned to keys following a uniform distribution. Keys are chosen uniformly at random from the key space, and values' size follows a normal distribution with a mean value of 5 MB and a standard deviation of 1 MB (yielding a total of around 250 GB). The maximum load threshold was set to 1.75. The group maintenance frequency was set to one second, with a probability of 10%. The periodic group size check was executed with a frequency uniformly distributed between two and four seconds. In turn, the periodic relocation of peers is checked every 20 seconds, and the relocation cool down period is also 20 seconds.

The overlay was initialized by having peers join the system one at a time. After a stabilization period, churn was induced during a period of 60 simulation cycles. Every other cycle during the churn period, c peers are removed simultaneously. When the churn period is over, another stabilization period is executed, and the simulation halts. As to reduce the number of experiments to a practical amount, we only used three churn rates from Table 5.2, namely 30% (low churn), 60% (medium churn), and 90% (high churn). We also use two different scenarios: one where peers leave the system and no new peers enter (that we called *exit-only*); and another scenario where peers leave the system and the same amount of new peers join the overlay (that we called *enter-exit*).

Groups are divided into two sets—hot and cold—, defined by a distribution ratio set to 50% (i.e., both sets have the same number of groups). Peers in the hot set have a probability ϵ of being chosen to leave the system (i.e., churn), while peers in the cold set have the complementary probability (i.e., $1 - \epsilon$). In these experiments, we set $\epsilon = 0.8$.

Here, we refer to the same configurations used in §5.5.2: **No PPR** (NPPR in the plots) - peer relocation disabled; **Push** - peer relocation using only push requests; **Pull** - peer relocation using only pull requests; and **Full PPR** (FPPR in the plots) - peer relocation fully enabled (i.e., using both push and pull requests). This study was conducted using the PeerSim simulator [180] and its event-driven engine. All results are averages extracted from 20 independent executions for each data point, and all the plots depict data collected from the start of the churn period until the end of the simulation.

Table A.1 shows the group size parameters used in this study. First, we select *four* peers as the minimum group size for all experiments (i.e., $l = 4$), which we argue is a reasonable and safe minimum value for many churn scenarios. Next, we select several maximum group size thresholds (i.e., h), in order to assess how the overlay behaves with increasing group sizes. For this, we select four main sizes, namely 8, 16, 32, and 64 peers. Additionally, we select an extra maximum size: 11. This is an intermediate size between the two previous smaller ones. It is an odd number because Parsley includes the limits in the allowed sizes, i.e., topology changes are only carried out if the current group size is strictly greater or lower than the limits. Thus, by having an odd number as the maximum limit, it means that the resulting groups after a split will have exactly the same size (because the number of peers will be even). Then, we experiment with various soft limit thresholds (i.e., l' and h'). We call *delta* to the difference between the soft and hard limits (i.e., $l' - l$ and $h' - h$). The tables' top row shows the delta value for each column.

Table A.1: Group size parameters in Parsley, varying soft limits amplitude.

(a) Size extra small.

Δ	0	1	2
l	4	4	4
l'	4	5	6
h'	8	7	6
h	8	8	8

(b) Size small.

Δ	0	1	2	3
l	4	4	4	4
l'	4	5	6	7
h'	11	10	9	8
h	11	11	11	11

(c) Size medium.

Δ	0	1	2	4	6
l	4	4	4	4	4
l'	4	5	6	8	10
h'	16	15	14	12	10
h	16	16	16	16	16

(d) Size large.

Δ	0	1	2	4	6	8	10	12	14
l	4	4	4	4	4	4	4	4	4
l'	4	5	6	8	10	12	14	16	18
h'	32	31	30	28	26	24	22	20	18
h	32	32	32	32	32	32	32	32	32

(e) Size extra large.

Δ	0	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
l	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
l'	4	5	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34
h'	64	63	62	60	58	56	54	52	50	48	46	44	42	40	38	36	34
h	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64

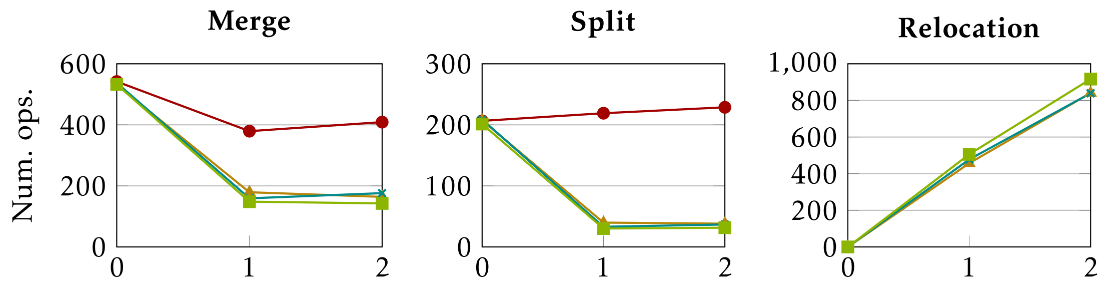
Since the maximum group sizes are even numbers, it allows us to experiment with deltas ranging from zero (disabling peer relocation completely) to the maximum being equal for both limits (i.e., $l' = h'$)—turning the desired group size interval into a single value.

A.2 Topology Operations and Data Transfers

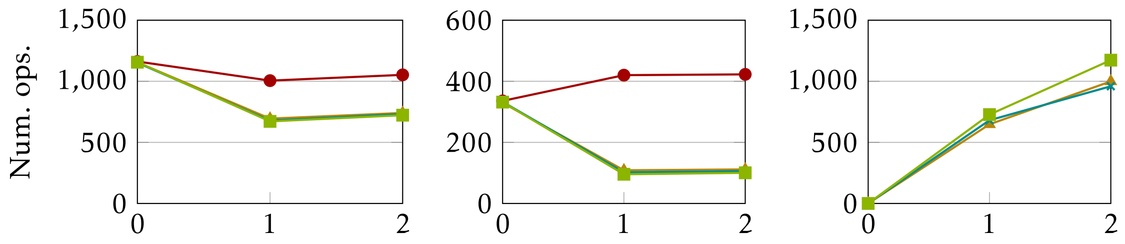
In this section, we analyze the impact of the different group sizes and deltas in the amount of topology operations executed (i.e., merges, splits, and peer relocations), and also in the amount of data objects transferred (in GB) due to these operations.

Size Extra Small: 4–8. This is the smallest size we experiment with, and with such a small size it is only possible to analyze three different deltas: 0, 1, and 2.

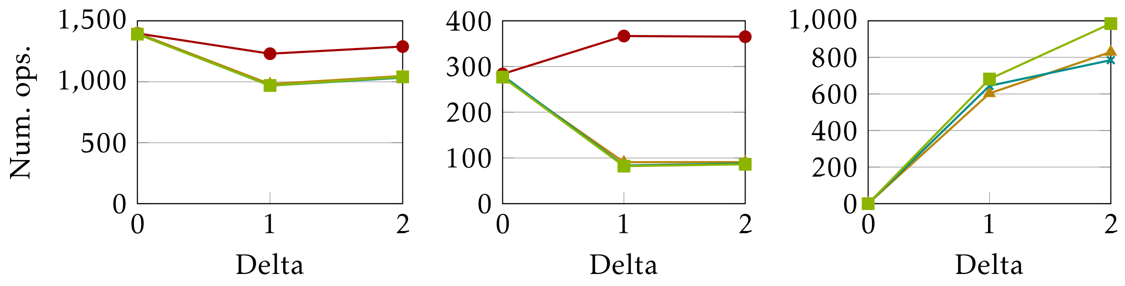
Figure A.1 depicts the amount of topology operations executed during the simulation, for the *exit-only* scenario. Since peers that leave the overlay are not replaced by new ones, increasing the amount of churn leads to a decrease in the overlay size. In the end, this results in the number of executed operations following the increase in churn (mainly merges), because groups have to accommodate those changes as more peers leave the system. Nonetheless, from delta 0 to 1, we see the largest decrease in both merge and split operations for the configurations with peer relocation. Here, even the NPPR configuration



(a) 30% churn.



(b) 60% churn.

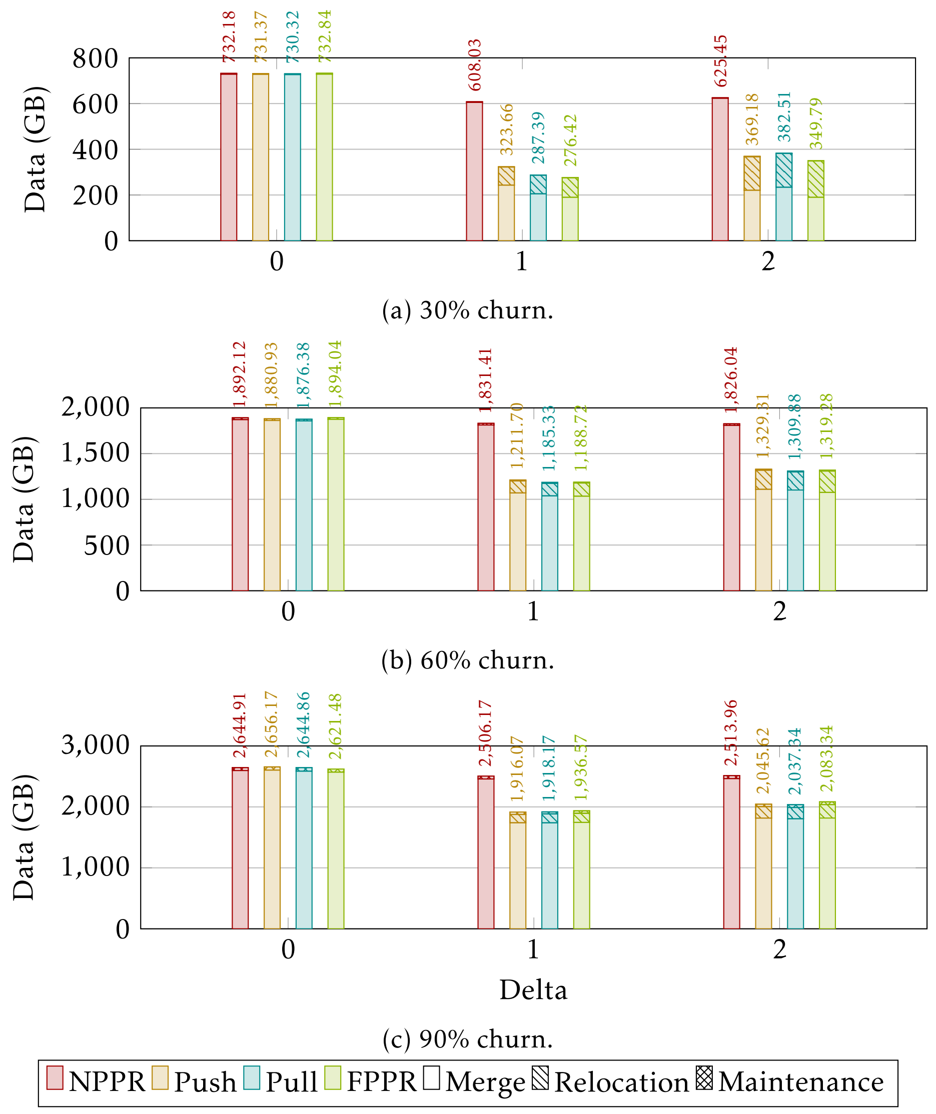


(c) 90% churn.

● NPPR ▲ Push × Pull ■ FPPR

Figure A.1: Exit-only topology operations with group size XS (4–8) in Parsley.

decreases the amount of merges, but (slightly) increases splits. The merges decrease because with delta greater than zero (specially with delta 1), the average size of the groups generated in the overlay bootstrap tends to be larger, thus giving rise to larger groups that are more robust to churn (see the new peer acceptance logic in §5.3.3.2). In turn, splits increase because, as explained in §5.3.3.4, additionally some of them start to happen due to groups being overloaded. Except for NPPR, peer relocations are an additional help to further lower the amount of these operations. Next, from delta 1 to 2, the difference is negligible (in fact, merges and splits increase slightly). With 30% churn, we can see a minor difference among the configurations with peer relocation (with FPPR achieving slightly less merges and splits, and more relocations), but that fades as churn increases. Since there are only peers leaving the overlay, for large values of churn, the only way to deal with this is to execute topology changes. Also, naturally, as the delta size increases, peers have more freedom to relocate, thus the number of relocation operations increases with the delta size. Since FPPR employs both push and pull requests, it presents more opportunities for peer relocations, thus it is the one that executes more relocations



(a) 30% churn.

(b) 60% churn.

(c) 90% churn.

Figure A.2: Exit-only data transfers with group size XS (4–8) in PARSLEY.

from the three. Yet, a relevant observation is that from delta 1 to 2, relocations double while merges and splits stay practically the same. This means that the bigger freedom peers enjoy with a larger delta may also cause many unnecessary relocations.

In Figure A.2, we can see the amount of data transfers resulting from merge, relocation, and group maintenance operations, for the *exit-only* scenario. First, we can see that with delta 0, all configurations behave similarly, since there are no relocations (which can also be seen in the previous figure). Also, it is clear that data transfers caused by group maintenance are a very tiny part of the overall transfers, being the total dominated by the other two parts (i.e., merges and relocations). However, with delta 1 or 2, all the configurations with peer relocation manage to require much less data transfers than NPPR, by greatly reducing the amount of transfers due to merges. With increasing amounts of churn, more peers leave the overlay, thus, in the end, there is no other possibility than to merge (with some relocations along the way). That is why increasing the delta size is

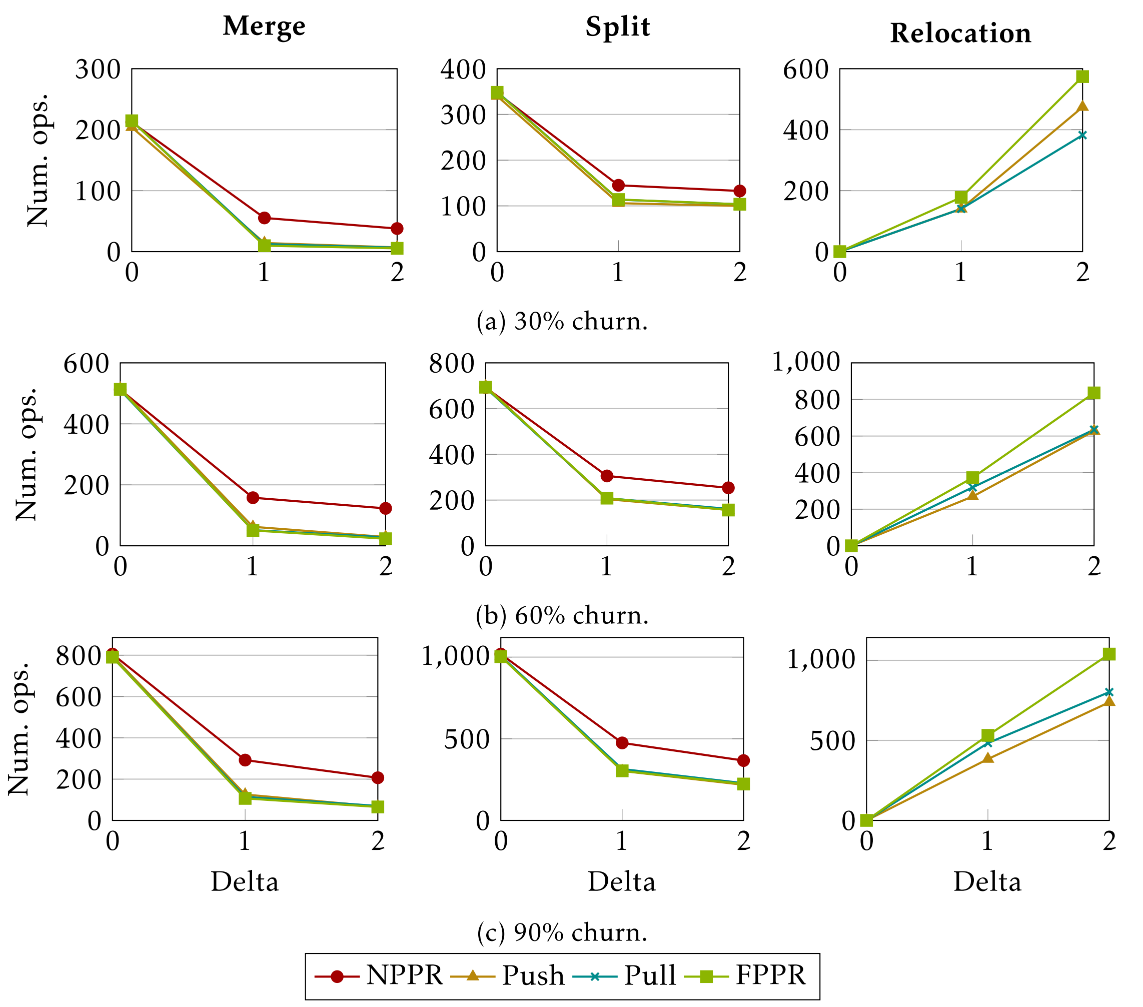
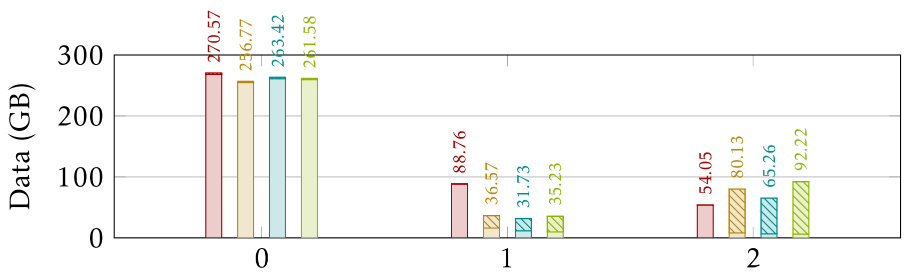


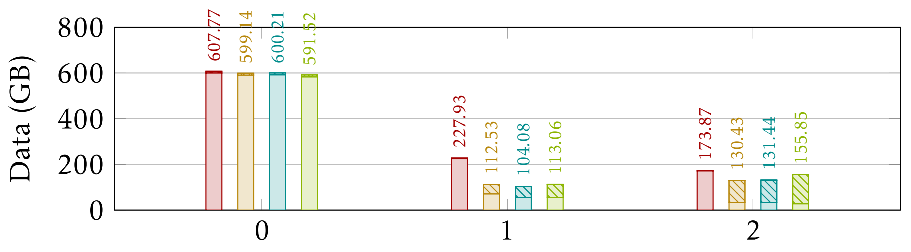
Figure A.3: Enter-exit topology operations with group size XS (4–8) in Parsley.

unable to further reduce the amount of data transfers, only enabling more relocations. In fact, from delta 1 to 2, the total data transfers increase slightly. Additionally, with increasing churn, the difference between NPPR and the other configurations becomes less evident, because groups get smaller and relocation opportunities diminish.

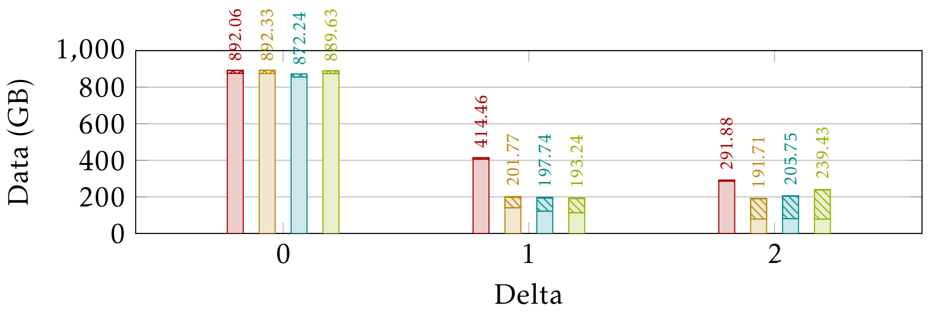
Figure A.3 depicts the amount of topology operations executed during the simulation, for the *enter-exit* scenario. Similarly to the *exit-only* scenario, the number of executed operations follows the increase in churn, i.e., the more churn is imposed on the overlay, the more topology operations are required in order to accommodate those transient changes. However, here, the absolute values are much smaller for merge operations. Since peers enter the overlay as others leave, they end up filling the voids. Thus, these operations are needed to accommodate the rapid changes in the network, but by a small amount when compared to the *exit-only* scenario. NPPR executes considerably more merge and split operations than any configuration with peer relocation, across all churn values. Once again, from delta 0 to 1, we see the largest decrease in merge (and split) operations for all configurations (even with NPPR). This decrease can be explained in part due to the same



(a) 30% churn.



(b) 60% churn.



(c) 90% churn.

□ NPPR □ Push □ Pull □ FPPR □ Merge ▧ Relocation ▩ Maintenance

Figure A.4: Enter-exit data transfers with group size XS (4–8) in Parsley.

reason described in the previous scenario (i.e., larger average group size with delta 1), and also adding to the fact that peers enter the overlay. In turn, from delta 1 to 2, the number of merges and splits decreases by a very small amount, led by the added freedom for relocations. However, the number of relocations increases linearly with the delta value. Here also, FPPR executes more relocations than any of the other configurations, since it employs both push and pull techniques.

In Figure A.4, we can see the amount of data transfers resulting from merge, relocation, and group maintenance operations, for the *enter-exit* scenario. Since there are no relocations with delta 0, all configurations behave identically. However, data transfers decrease sharply with delta 1, and all configurations with peer relocation are able to reduce the amount of transfers to half that of NPPR (with FPPR achieving the lowest of the three). By increasing the delta value to 2, it allows more peer relocations than required, and thus the relocation data transfers completely dominate the total amount (which can be seen clearly in Figure A.4a, for instance). This is the reason that with delta 2, FPPR

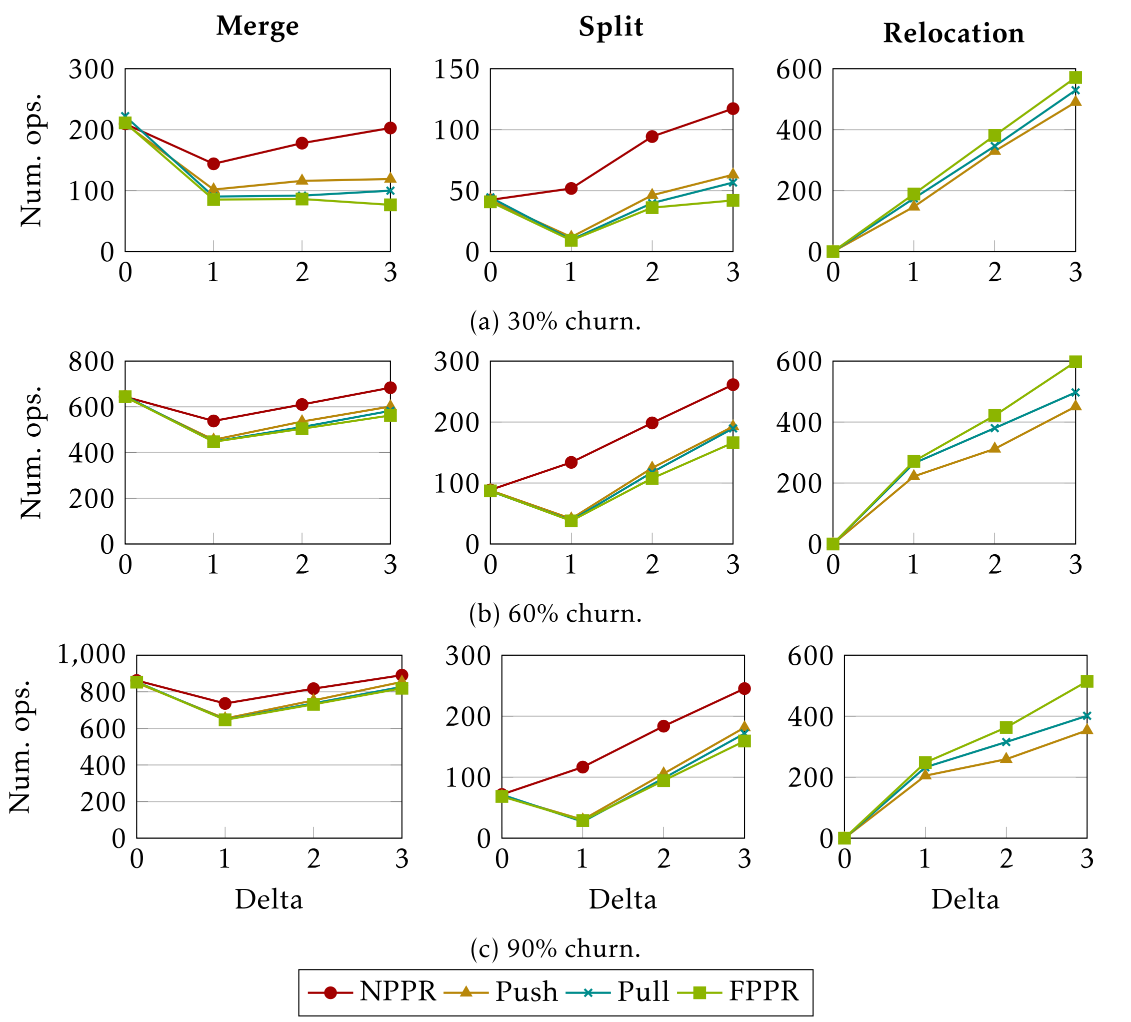
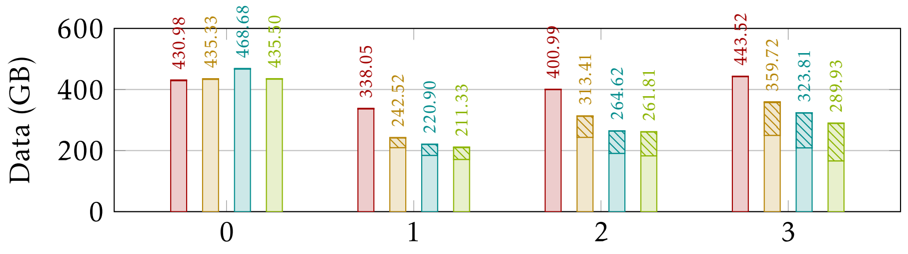


Figure A.5: Exit-only topology operations with group size S (4–11) in Parsley.

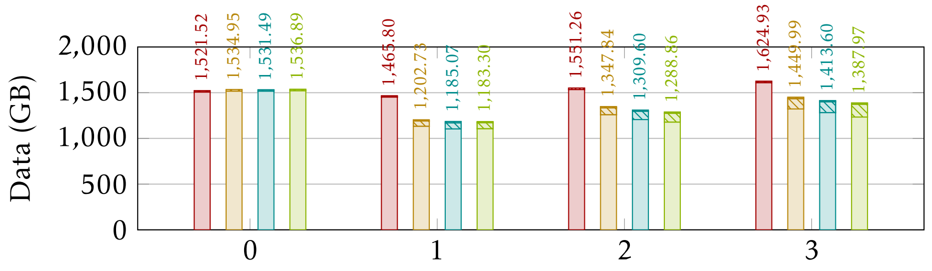
requires more data transfers than the other two configurations with peer relocation—it enables more unnecessary (and unfruitful) freedom.

Size Small: 4–11. This is the only group size range with an odd maximum limit, thus allowing the two groups resulting from a split to be exactly the same size.

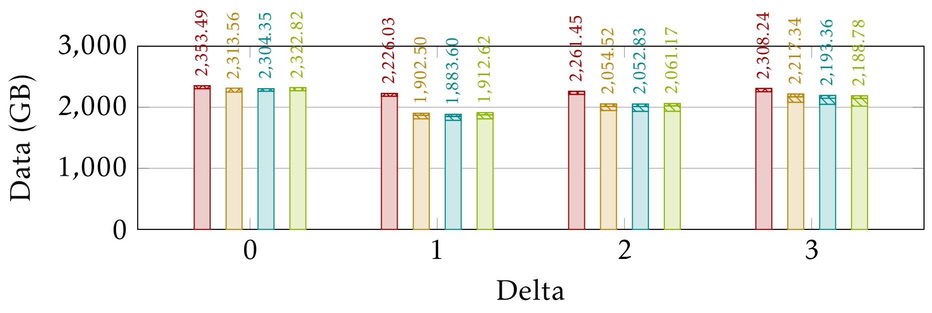
Figure A.5 depicts the amount of topology operations executed during the simulation, for the *exit-only* scenario. Once more, from delta 0 to 1, we see the largest decrease in both merge and split operations, mainly for the configurations with peer relocation. Then, this is followed by an increase in both metrics as the delta value increases, also accompanied by a sub-linear increase in the number of peer relocations (very similar for all the churn values). The decrease in merges with delta 1 can be explained by the same reason as in the previous group size range. With delta 1, the average size of the groups generated in the overlay bootstrap is significantly larger than with delta 0, thus groups go into the churn period better equipped in case they lose peers. However, with delta 2 onward, groups' size starts to approach the middle of the interval defined by the parameters (in



(a) 30% churn.



(b) 60% churn.



(c) 90% churn.

🟥 NPPR 🟨 Push 🟦 Pull 🟩 FPPR □ Merge ▨ Relocation ⊠ Maintenance

Figure A.6: Exit-only data transfers with group size S (4–11) in Parsley.

this case, 8), i.e., the average size starts to decrease slowly (and the standard deviation also). That is why the number of merges reverses and starts to (slowly) grow. In the case of the configurations with peer relocation, they require less merges because peer relocations are able to balance that. Regarding splits, they increase with NPPR, and also with delta 2 onward for the other configurations. This happens because, as peers leave the overlay, groups keep their data and start to become overloaded. In the end, the majority of the splits are due to overload and not group size. From the previous group size range, there is also a decrease in the amount of relocations. This can be explained by the fact that larger groups are more robust to churn, thus requiring less relocations.

In Figure A.6, we can see the amount of data transfers resulting from merge, relocation, and group maintenance operations, for the *exit-only* scenario. Here, we can see that as churn increases, the difference between NPPR and the configurations with peer relocation decreases. This happens because no peers entering the overlay means that groups shrink and there is no other alternative but to merge. This difference among configurations

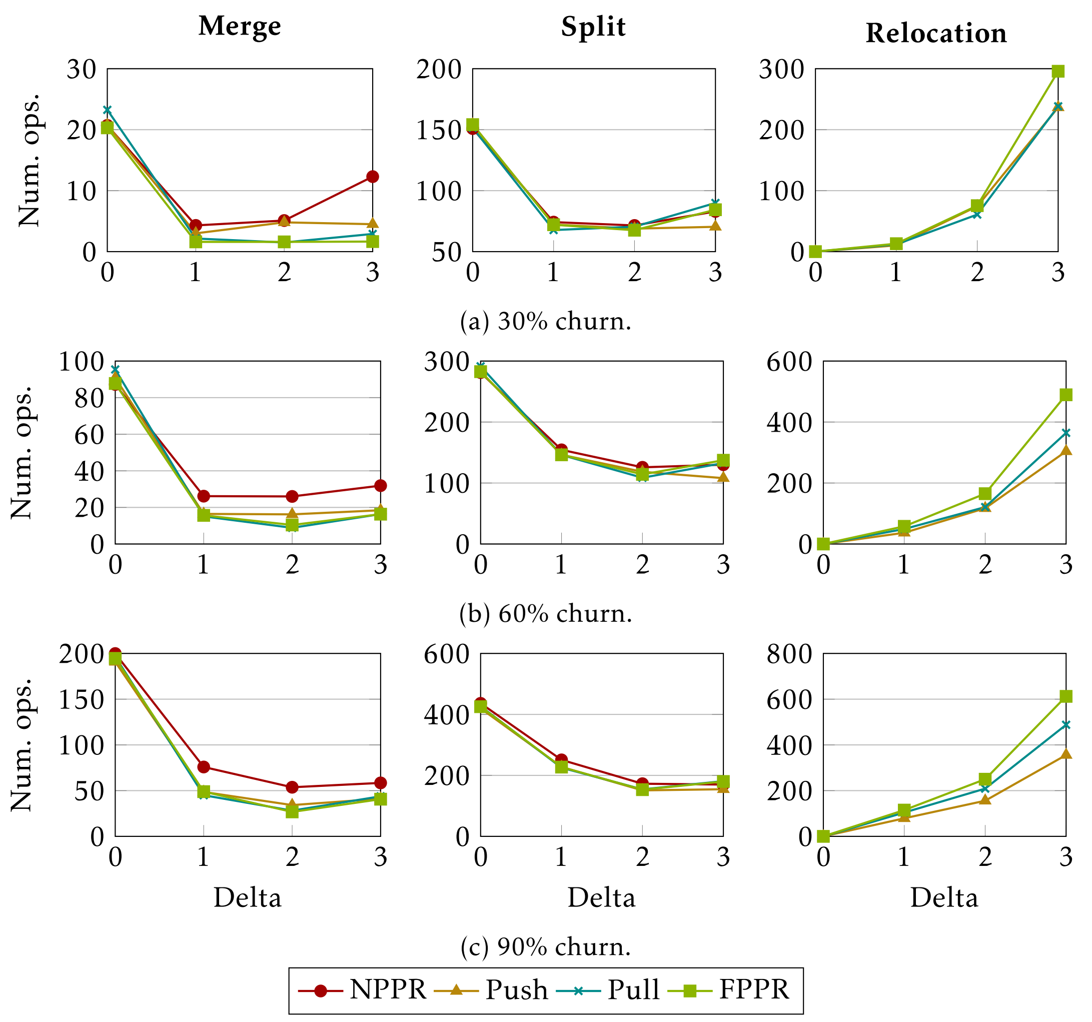
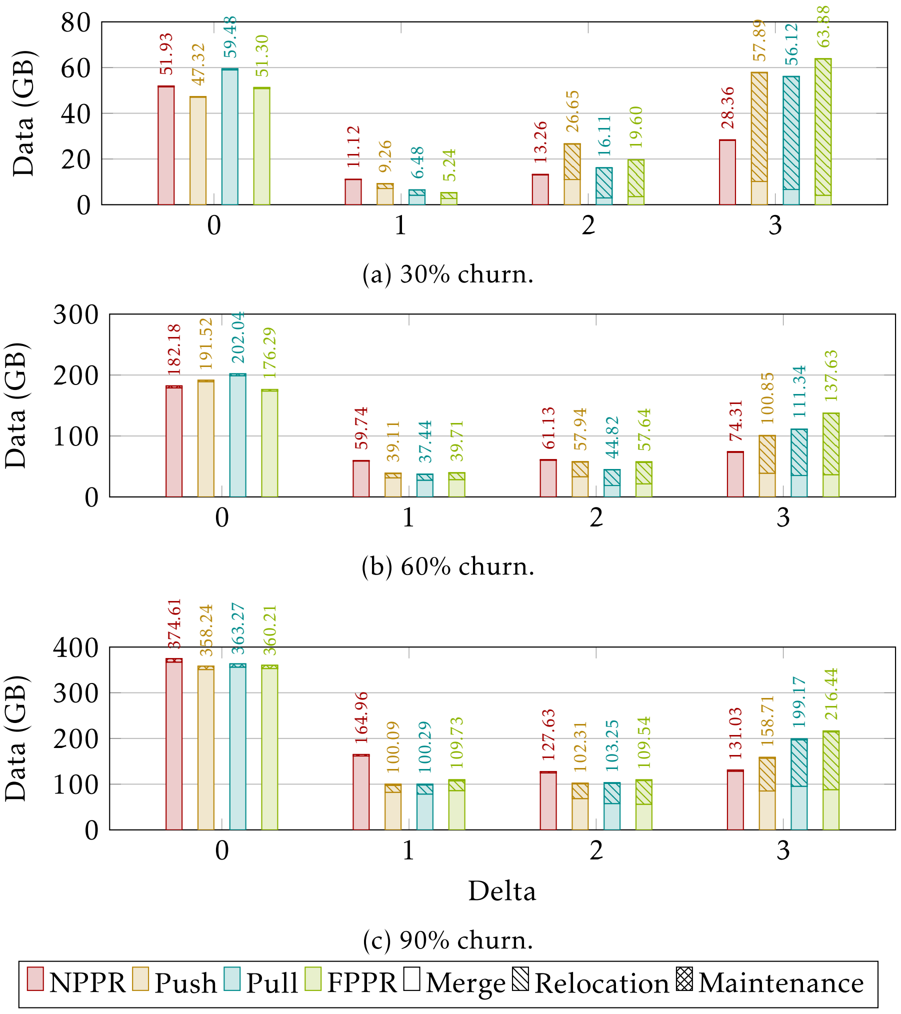


Figure A.7: Enter-exit topology operations with group size S (4–11) in PARSLEY.

also decreases as delta size grows. Since no peers enter the overlay, in the scenario with less churn, the configurations with peer relocation require much more data transfers due to relocations. Then, as churn increases and relocation opportunities decrease, the amount of relocation data transfers also decreases. The figure also shows that all the configurations with peer relocation behave similarly, with FPPR achieving slightly lower overall data transfers.

Figure A.7 depicts the amount of topology operations executed during the simulation, for the *enter-exit* scenario. Here, since new peers replace the leaving ones, with low churn, all the configurations behave similarly (overlapping for the most part in the plots). However, as churn increases, the configurations with peer relocation start to reduce the number of merge (and also slightly split) operations when compared with NPPR, offset by the performed relocations. Also, the largest decrease in merge (and split) operations is noticed when delta goes from 0 to 1. As the delta increases, the variation among the configurations starts to become less visible. Naturally, the number of relocations grows with the delta value, as peers have more freedom to relocate between groups.



(a) 30% churn.

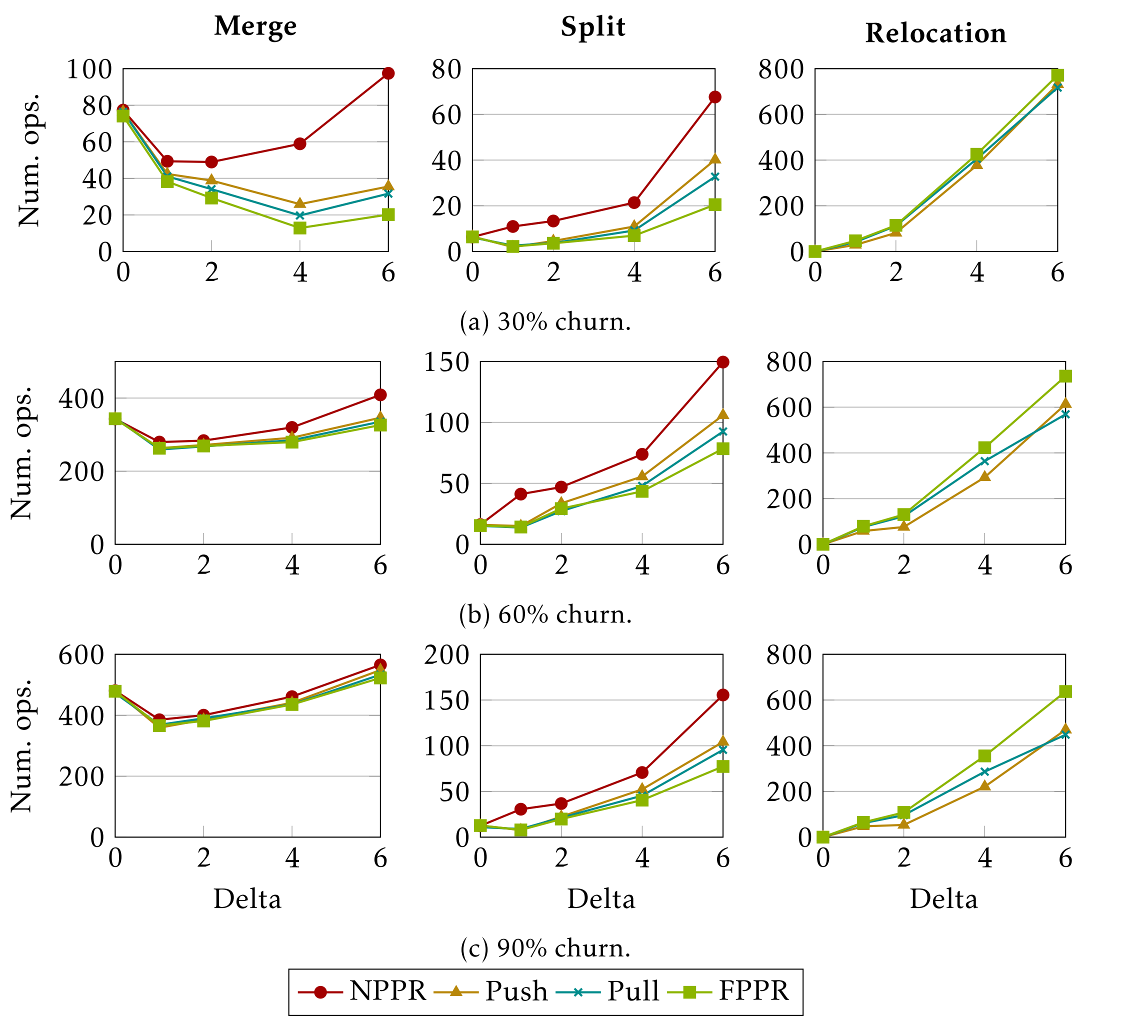
(b) 60% churn.

(c) 90% churn.

Figure A.8: Enter-exit data transfers with group size S (4–11) in Parsley.

In Figure A.8, we can see the amount of data transfers for the *enter-exit* scenario. With delta 1 and for all churn values, all configurations manage to sharply reduce the overall data transfers, and specially the ones with peer relocation achieve the lowest values. Naturally, as churn grows, more data transfers are required. Still, as the delta grows, peers have more unnecessary freedom, and start to relocate more, reaching a point where the transfers due to relocation surpass that of merges by a great margin. Here, the configurations with peer relocation achieve a reduced amount of data transfers due to merge. However, for instance, with delta 3, peer relocations completely dominate the data transfers, and causes the total to exceed that of NPPR by a considerable amount.

Size Medium: 4–16. From this range onward, groups start to have a considerable size. Here, the maximum limit used in this group size range doubles that of the first one, allowing us to experiment with five different deltas. With delta 0, group size can vary between four and 16 (i.e., the hard limits). In turn, with delta 6, groups will try to stay close to the middle of the range, i.e., with 10 peers.



(a) 30% churn.

(b) 60% churn.

(c) 90% churn.

Figure A.9: Exit-only topology operations with group size M (4–16) in Parsley.

Figure A.9 depicts the amount of topology operations executed during the simulation, for the *exit-only* scenario. First, both the amount of merge and split operations is cut in half from the previous group size range, continuing to showcase the natural trend that bigger groups are more robust to churn. Also, as churn increases, NPPR gets closer to the configurations with peer relocation (mainly regarding merges), because there are only peers exiting the overlay, and there is not much to do than merge. Nonetheless, with 30% churn, the number of merges for the configurations with peer relocation manages to decrease with increasing delta values, led by the freedom of peer relocations. At the same time, for NPPR, merges grow with the delta size (from delta 1 onward). Here, the amount of merges grows for the same reason as explained before. The average group size grows with delta 1, but then it starts to decrease, approaching the middle of the parameterized range (also with a smaller standard deviation). This also happens for the configurations with peer relocation and high churn values. In this case, the high amount of peer relocations ceases to have a beneficial effect, since the number of peers leaving the overlay makes that the only viable option is to merge. Regarding splits, they

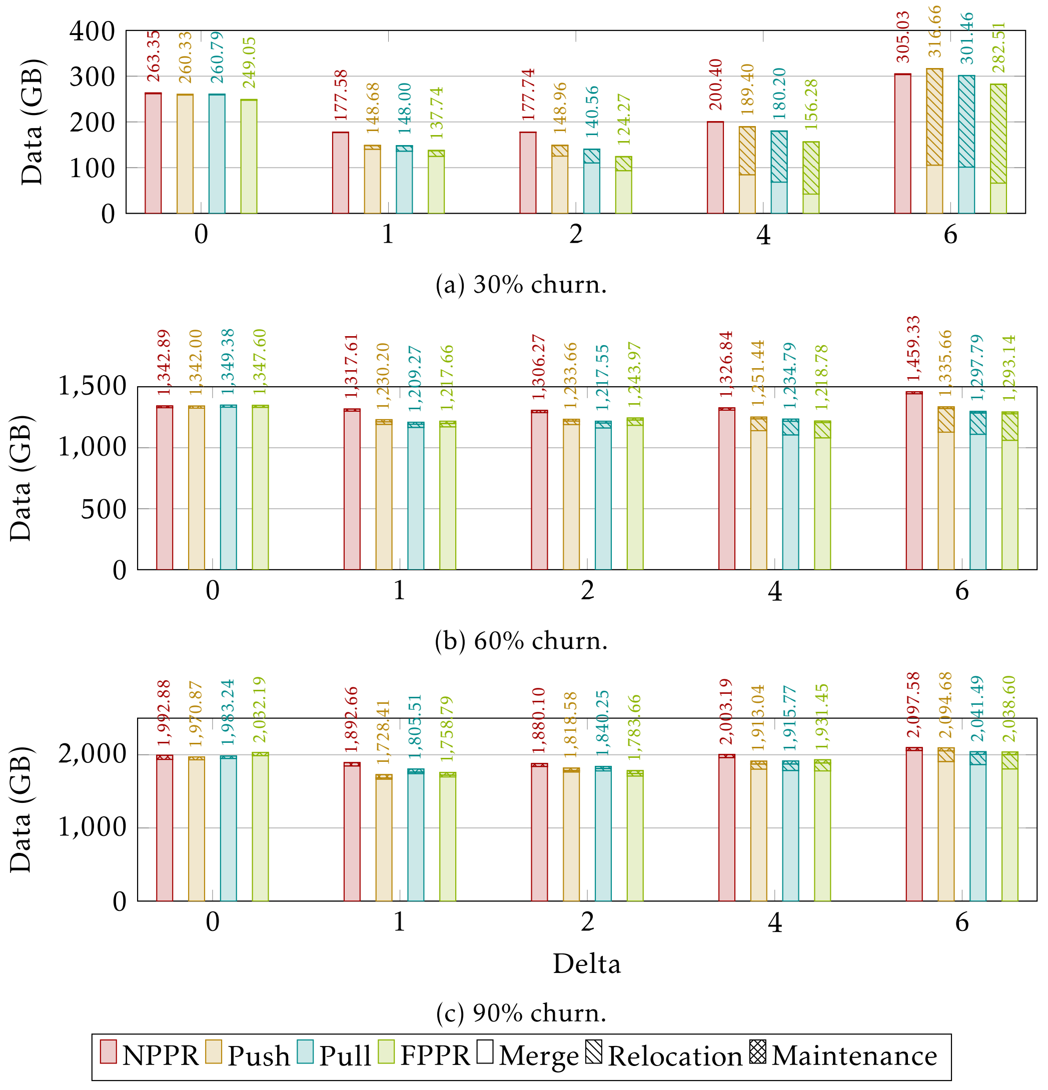
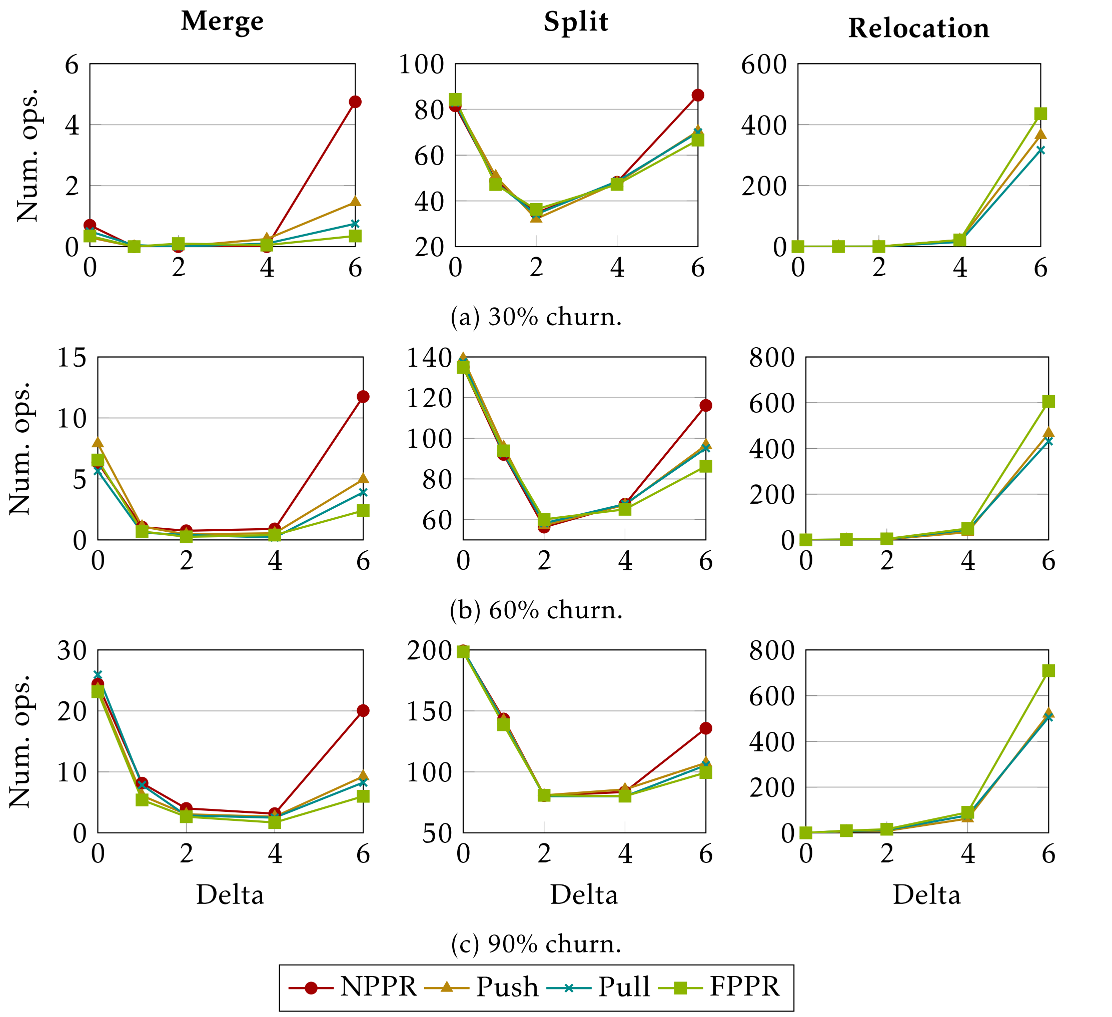


Figure A.10: Exit-only data transfers with group size M (4–16) in PARSLEY.

increase for all configurations, as the delta size grows. However, NPPR always requires considerably more split operations. Splits grow due to the same reason as mentioned in the previous group size ranges. As peers leave the overlay, groups become overloaded and split according to the defined logic (§5.3.3.4).

In Figure A.10, we can see the amount of data transfers resulting from merge, relocation, and group maintenance operations, for the *exit-only* scenario. Overall, in these plots, there is little variation. As mentioned before, except with 30% churn, NPPR is very similar to the configurations with peer relocation. This is mainly regarding merges, but also somewhat with splits. Thus, it is natural that all configurations have similar results for data transfers. That is exactly what we can see in the plots with high churn values, where the configurations with peer relocation manage to require a little less data transfers across all delta sizes. However, with 30% churn, we can see some movement in the plots, as the delta size increases. As in the previous figure, here, the largest decrease in data



(a) 30% churn.

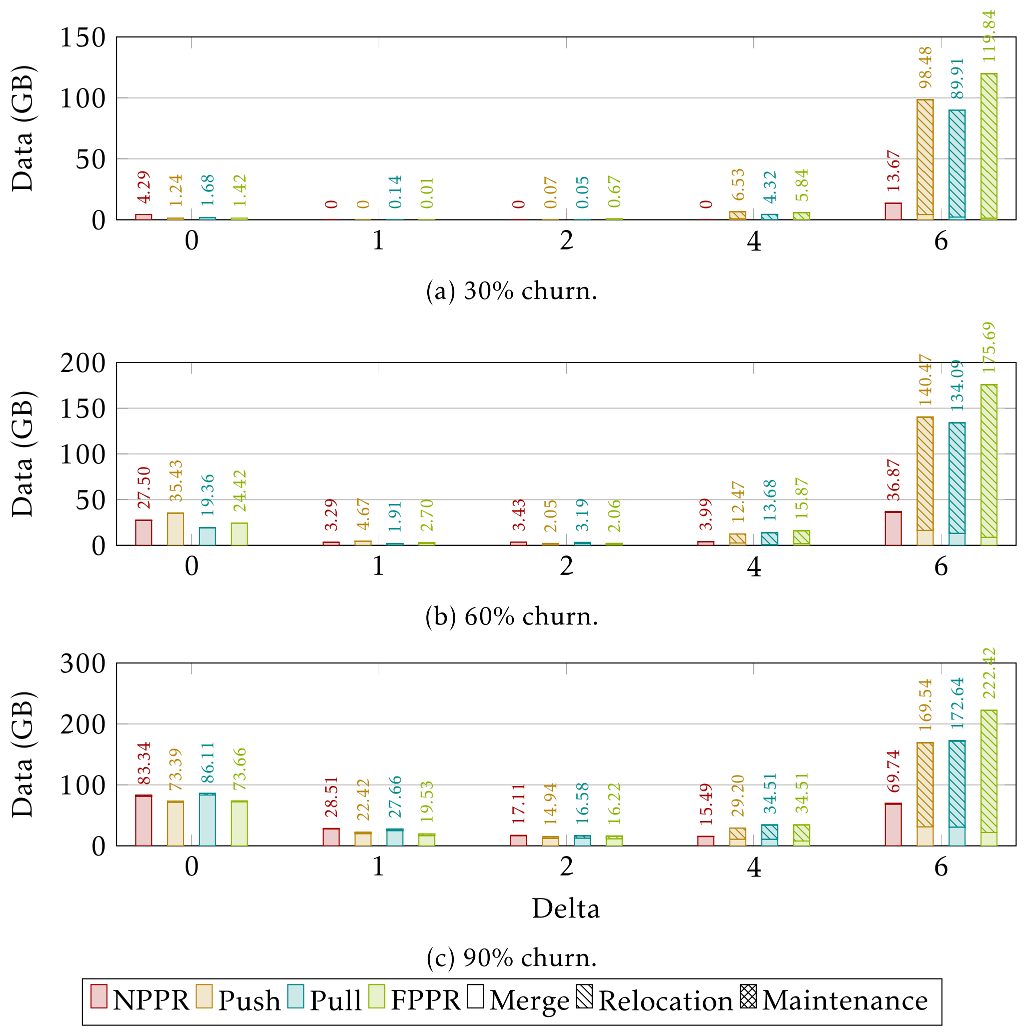
(b) 60% churn.

(c) 90% churn.

Figure A.11: Enter-exit topology operations with group size M (4–16) in Parsley.

transfers happens when going from delta 0 to 1. From that point onward, data transfers start to increase, due to the increased freedom peers enjoy with high delta sizes. With delta 6, we have twice as many data transfers due to relocations than due to merges.

Figure A.11 depicts the amount of topology operations executed during the simulation, for the *enter-exit* scenario. With peers entering the overlay to substitute the ones leaving and with a considerable maximum group size, we start to see the number of merges becomes substantially smaller, even for large values of churn. Here, even NPPR behaves similarly to the configurations with peer relocation, varying very little among them. It only diverges notably with delta 6, and even so, the variation is not that significant because we are talking about really small absolute values. Again, the largest decrease is noticed when delta goes from 0 to 1. Now, there is a sharp drop in splits until delta 2, and only then starts to increase, with NPPR requiring a larger amount of splits. Here, until delta 2, the vast majority of the splits are due to group size. Yet, with delta 4, some splits start to happen due to overload, and with delta 6 there is a sharp increase, with more than a third of the splits being due to overload. This can be explained by the same



(a) 30% churn.

(b) 60% churn.

(c) 90% churn.

Figure A.12: Enter-exit data transfers with group size M (4–16) in Parsley.

reason as in previous group size ranges, since it can influence both merges and splits. In this scenario, with delta 1 and 2, the average size of the groups generated in the overlay bootstrap is significantly larger than with delta 0, thus there are more larger groups going into the churn period. In turn, with higher deltas, groups' size starts to decrease and approach the middle of the interval defined by the parameters (in this case, 10 peers). Regarding peer relocation, this operation is practically non-existent until delta 4. Only with delta 6, relocations sharply increase, given the enhanced freedom this delta allows. However, from the number of merges required in this scenario, this amount of relocations is completely unnecessary.

In Figure A.12, we can see the amount of data transfers resulting from merge, relocation, and group maintenance operations, for the *enter-exit* scenario. The churn impact can be seen perfectly in this figure, with data transfers being small compared to the previous group size ranges. We can also see that, in this scenario, peer relocations may not be advantageous in some cases. In fact, for large delta values, it becomes detrimental,

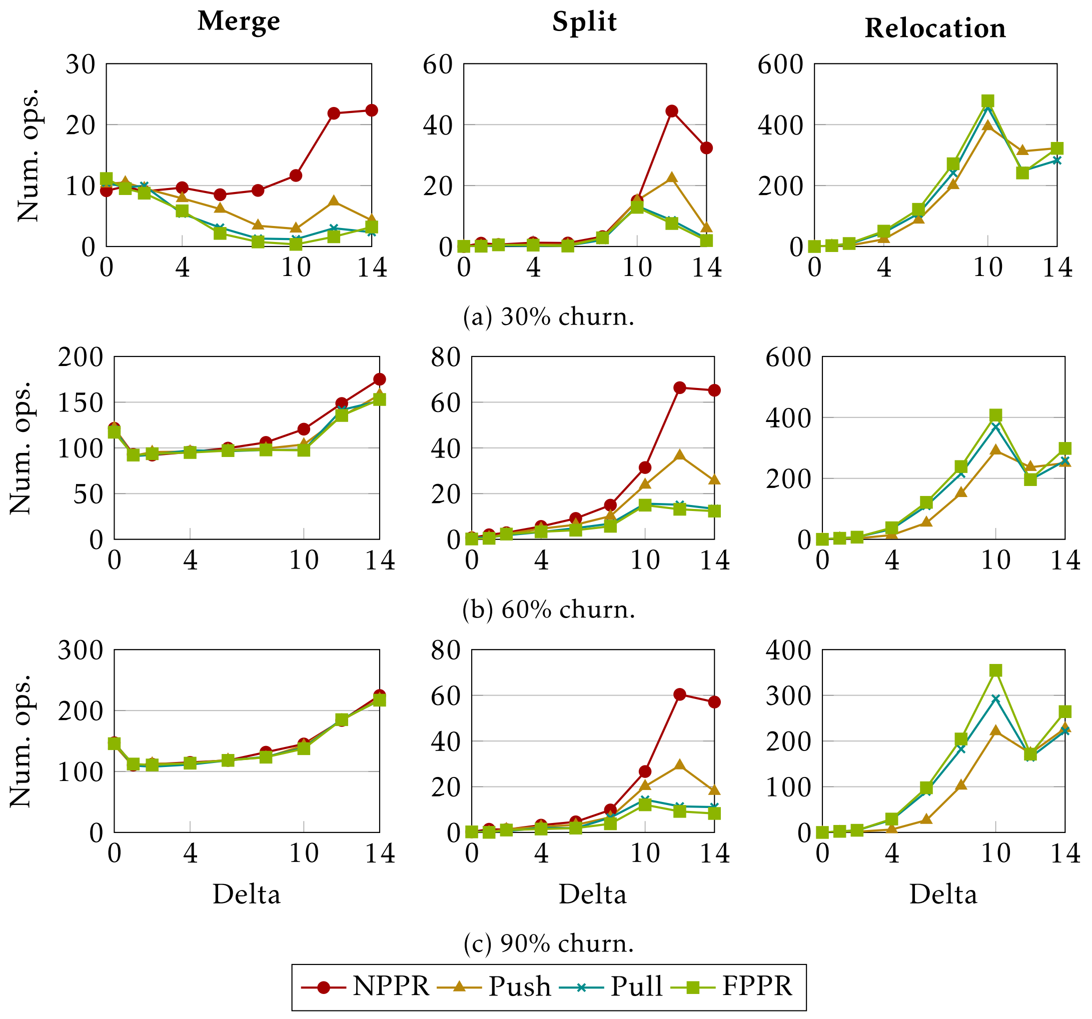
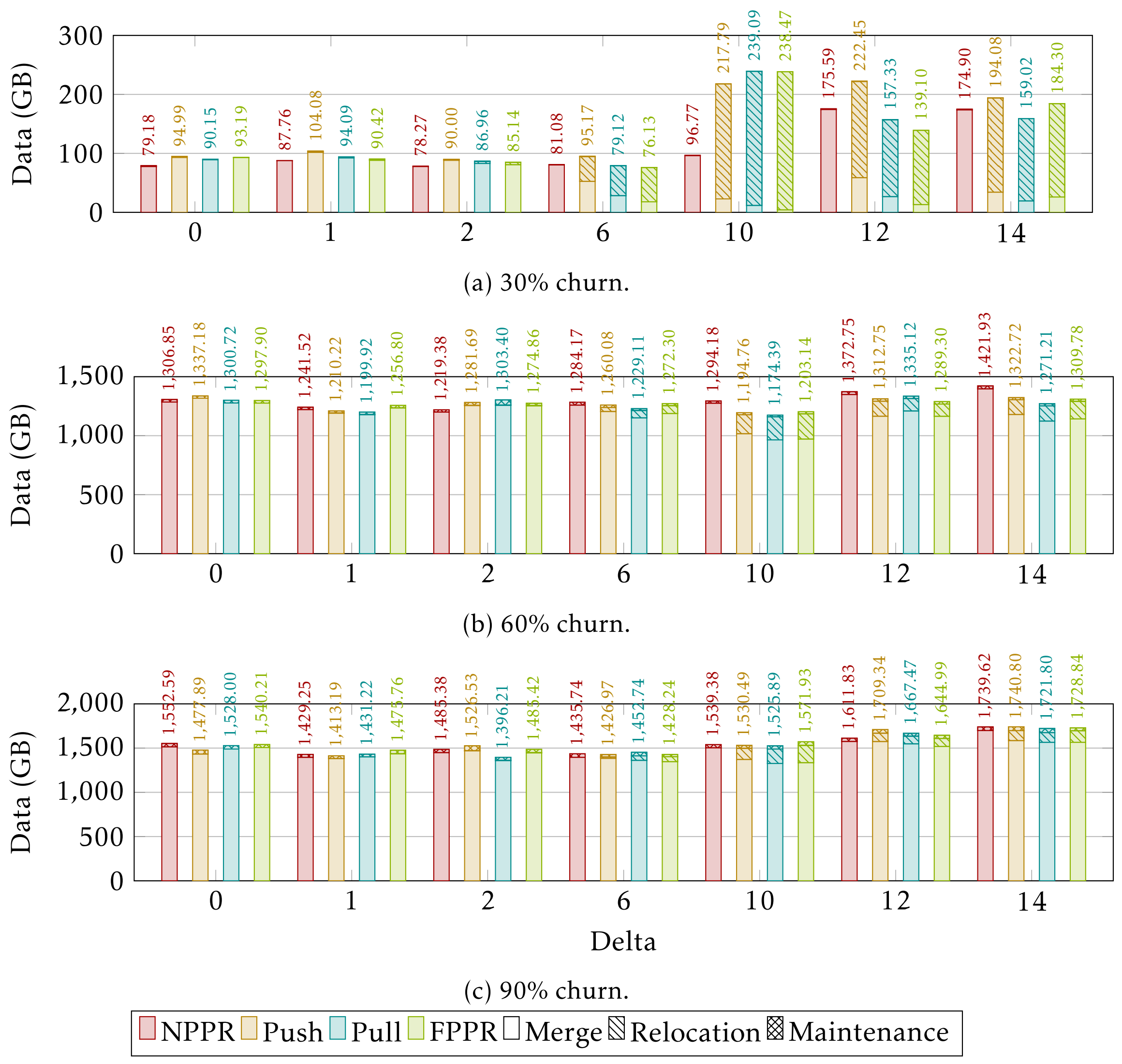


Figure A.13: Exit-only topology operations with group size L (4–32) in PARSLEY.

with peer relocations completely dominating the total data transfers. For instance, with delta 6, it effectively reduces the amount of data transferred due to merges, but at the cost of an excessive amount of (unnecessary) peer relocations.

Size Large: 4–32. This range allows groups to be quite large (thus we call this size large). It allows us to experiment with nine different deltas.

Figure A.13 depicts the amount of topology operations executed during the simulation, for the *exit-only* scenario. These plots are similar to the same scenario in the previous group size range, reducing their values by around half. With low churn (i.e., 30% churn), merges decrease until delta 10 for the configurations with peer relocation, and then stabilize. For NPPR, merges stay stable until delta 10, and then start to increase. Relocations end up compensating for the groups' smaller size with the delta increase—something that NPPR cannot. With high churn, all configurations behave similarly. They start with the largest decrease when delta goes from 0 to 1, stabilize until delta 10, and then start to increase. Since there are no peers entering the overlay, peer relocations cannot offset



(a) 30% churn.

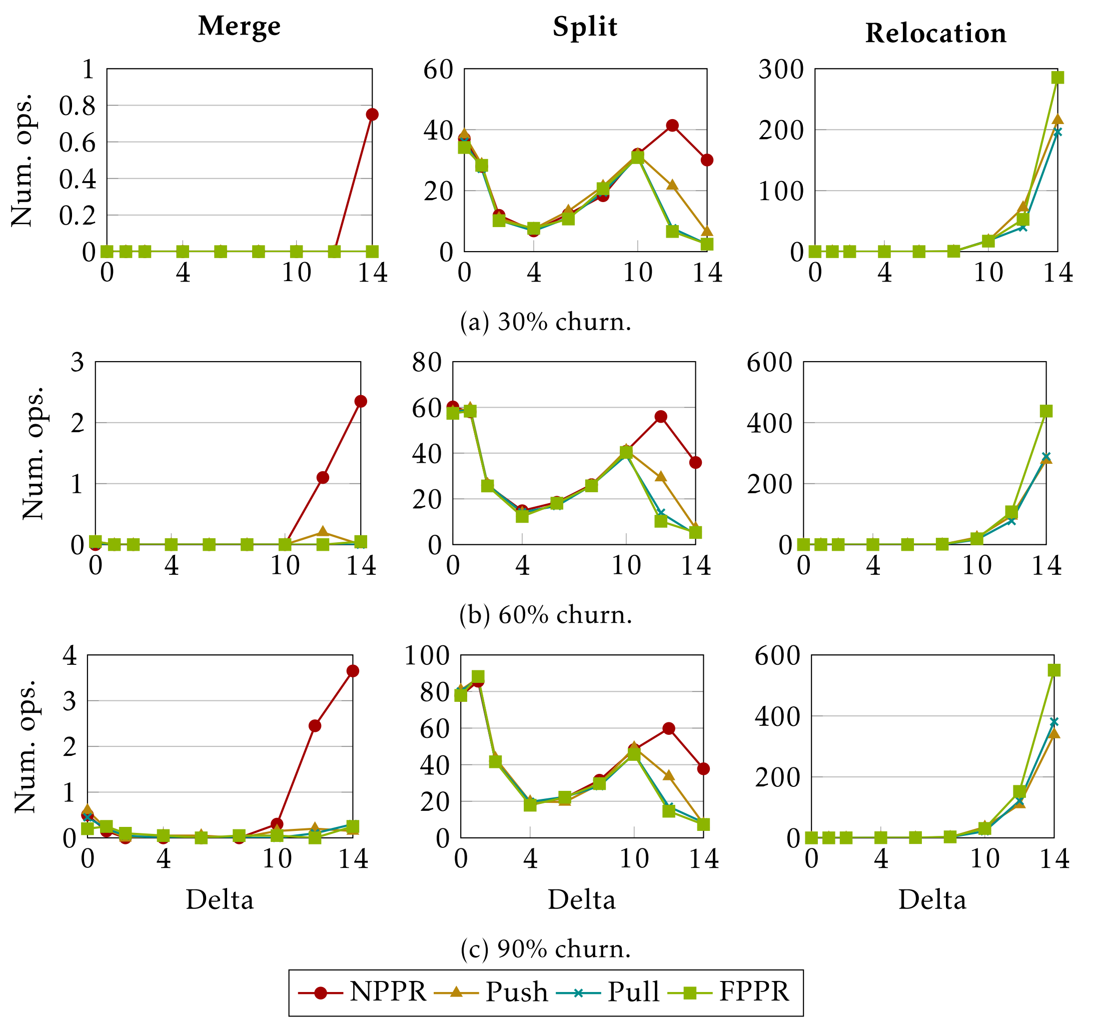
(b) 60% churn.

(c) 90% churn.

Figure A.14: Exit-only data transfers with group size L (4–32) in Parsley.

that with these levels of churn. Splits are almost non-existent until delta 8, but nearly all are due to overload. However, then they start to increase with the delta size, with NPPR growing more than the other configurations. Peer relocations grow until delta 10, where there is an inflection point, dropping almost half, to then increase again with delta 14. For all deltas, with FPPR, around two thirds of the relocations are due to pull requests.

In Figure A.14, we can see the amount of data transfers resulting from merge, relocation, and group maintenance operations, for the *exit-only* scenario. These plots reflect clearly the numbers in the previous figure. With low churn, there are few operations, thus the amount of data transfers is also reduced. Specially, we can see that for small deltas, all configurations require a small amount of data transfers. In turn, with large deltas, peer relocations completely dominate the transfers, as peers have a large degree of freedom to relocate (effectively too much). On the other hand, with high levels of churn, all configurations behave identically, with the ones with peer relocation requiring slightly



(a) 30% churn.

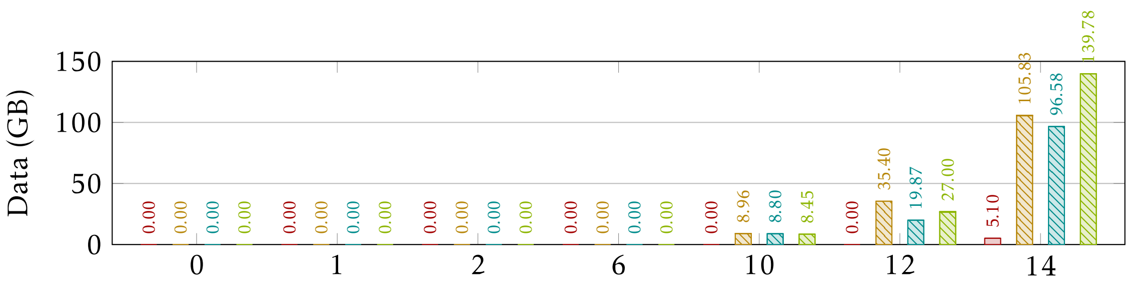
(b) 60% churn.

(c) 90% churn.

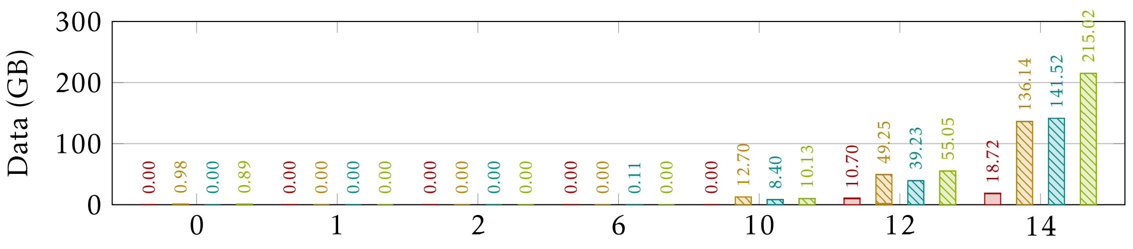
Figure A.15: Enter-exit topology operations with group size L (4–32) in Parsley.

less transfers. With high churn and peers only leaving the overlay, the system reaches a point where it has no other option than to merge groups. Notice that due to space and presentation concerns, these plots do not present the values for all the deltas. For completeness sake, we present a different plot with all the values (including the omitted ones) at the end of the appendix (see Figure A.25).

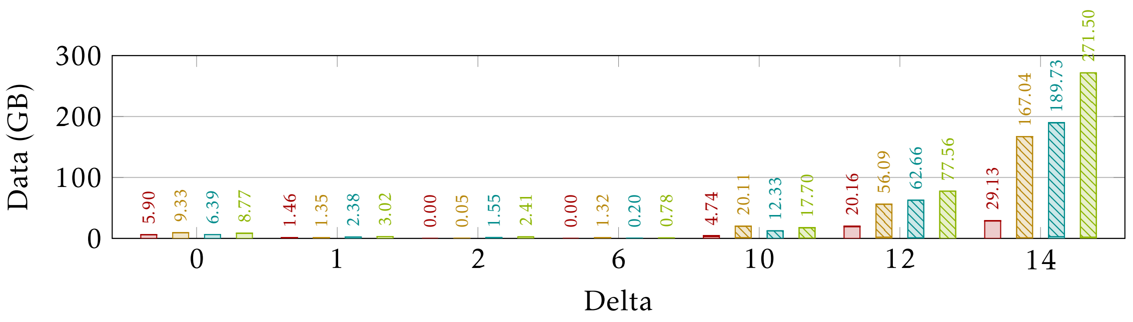
Figure A.15 depicts the amount of topology operations executed during the simulation, for the *enter-exit* scenario. Here, we see clearly the effects of large groups together with the fact that peers enter the overlay as others leave. First, for all levels of churn, the number of merge operations is negligible. Only NPPR requires a minute number of merges with very large deltas. Then, regarding splits, all the configurations overlap for the most part in the plots. The number of splits decreases until delta 4, then starts to increase. From delta 10, it sharply decreases (almost to zero) but only for the configurations with peer relocation. Still, for the configurations with peer relocation, with delta 10 and 12, around a third of the splits are due to group overload. In turn, for NPPR, with delta 10, around a third of the splits are due to group overload. Onward, these grow



(a) 30% churn.



(b) 60% churn.



(c) 90% churn.

▭ NPPR ▭ Push ▭ Pull ▭ FPPR ▭ Merge ▨ Relocation ▩ Maintenance

Figure A.16: Enter-exit data transfers with group size L (4–32) in PARSLEY.

to roughly two thirds. Because groups are large, peer relocations only start to happen with delta 10, and then increase rapidly with the delta size. However, with these large groups, since merges are not necessary, in the end, whatever the overlay does regarding peer relocations will always be somewhat counter-productive and wasteful.

In Figure A.16, we can see the amount of data transfers resulting from merge, relocation, and group maintenance operations, for the *enter-exit* scenario. Since there are only a minute amount of merges, the vast majority of the data transfers is due to peer relocations. In the plots, we can see that with small deltas, there are practically no need to transfer data around. However, with large deltas (i.e., delta 10 onward), since peers enjoy a large degree of freedom to relocate, we can see another proof that confirms what was previously mentioned—the peer relocations executed in this scenario are not beneficial to the overall system. In fact, data transfers due to peer relocations increase greatly with large deltas, but in a detrimental way. As in Figure A.14, these plots also do not present the values for all the deltas. Similarly, we show a complete plot with all the values at the

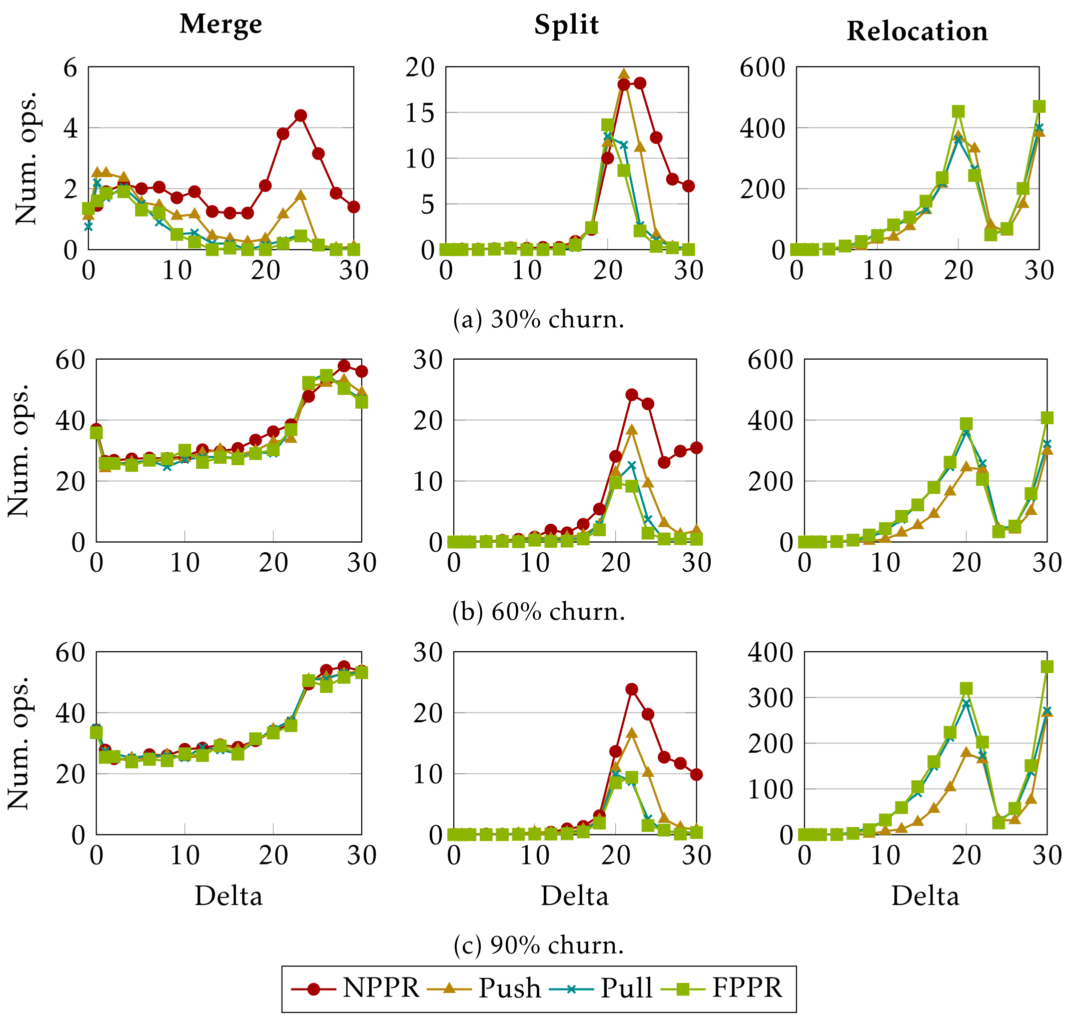


Figure A.17: Exit-only topology operations with group size XL (4–64) in Parsley.

end of the appendix (see Figure A.26).

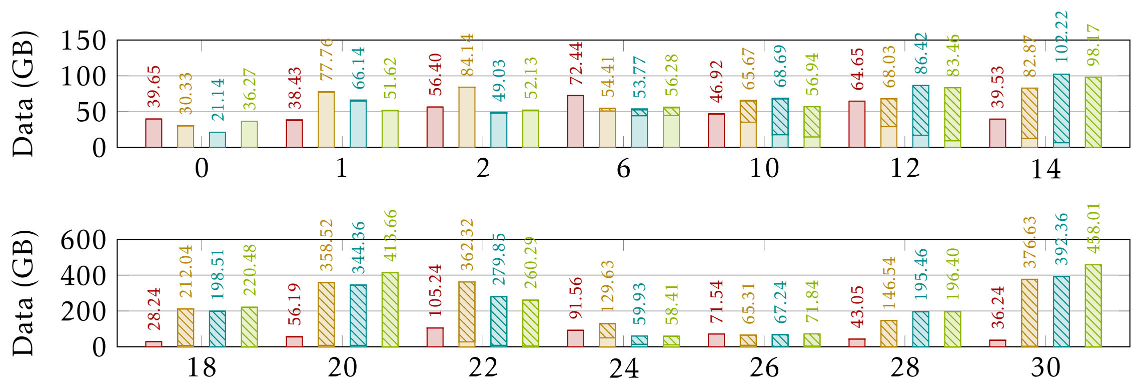
Size Extra Large: 4–64. This is the biggest maximum limit we use for group size, generating huge and robust groups—that is why we call it size extra large. It allows us to experiment with 17 different deltas.

Figure A.17 depicts the amount of topology operations executed during the simulation, for the *exit-only* scenario. Even in this scenario, where there are only peers leaving the overlay, merge operations start to become rare, as such big groups tolerate high churn more easily. Only with high levels of churn and large deltas, merges peak at around 60 operations. Once again, the largest decrease in the amount of merge operations happens when the delta goes from 0 to 1. Here, this decrease is mostly due to the increased groups' size, since there are almost no peer relocations taking place with this delta. Regarding split operations, they also happen sparingly, with a peak around delta 22, and practically all are due to group overload. Peer relocations start to occur at around delta 10, and gradually increase until delta 20. Its amount drops sharply, almost to zero, until delta 24, to then increase again to values around the previous maximum (thus, more rapidly). The majority of relocations are due to pull requests. Only with delta 30 this inverts, and two thirds of the relocations are due to push requests.

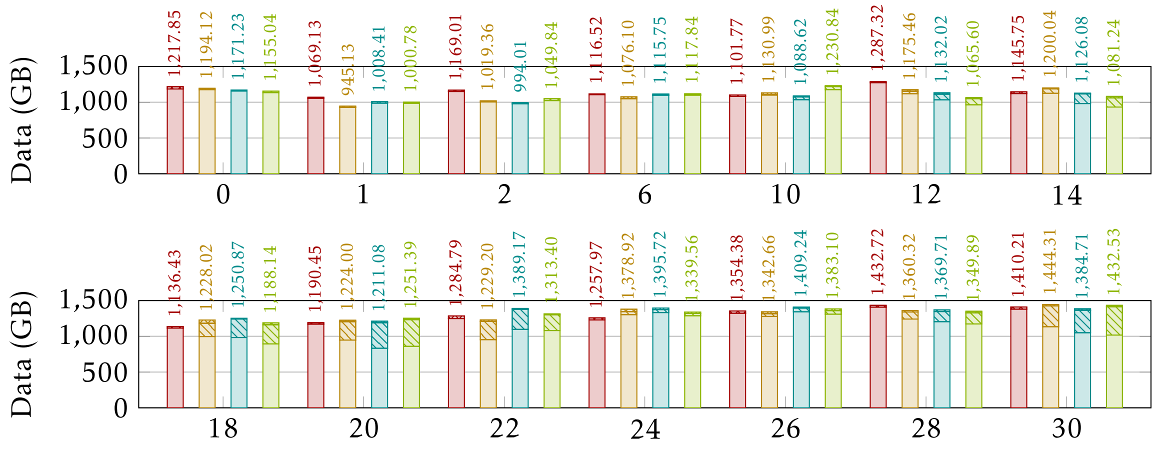
In Figure A.18, we can see the amount of data transfers resulting from merge, relocation, and group maintenance operations, for the *exit-only* scenario. Since the number of operations among all configurations does not differ much, they all present similar values regarding data transfers. This is most evident for high churn levels. In turn, with 30% churn, we can see that the configurations with peer relocation require much more data transfers, specially as the delta increases. This happens due to the increased freedom peer have as the delta size grows. However, as the number of merge operations reflects, these relocations are actually detrimental to the overall data transfers (and consequently to the system performance). Thus, for large deltas, the configurations with peer relocation end up requiring much more transfers than NPPR. Notice that due to space and presentation concerns, these plots do not present the values for all the deltas. For completeness sake, we present a different plot with all the values (including the omitted ones) at the end of the appendix (see Figure A.27).

Figure A.19 depicts the amount of topology operations executed during the simulation, for the *enter-exit* scenario. In this case, with such big groups and having peers entering the overlay to replace the leaving ones, merge operations are barely necessary. Since group are so big, they work as a dampener and can handle these levels of churn easily. Moreover, since peers enter the overlay by the same amount as those that leave, the overlay keeps its size stable. Nonetheless, splits still happen, although that in small amounts. For all churn levels, the amount of splits decreases until delta 10, to then start to spike around delta 20. From this spike, nearly all splits are due to group overload (instead of group size), and then start to decrease almost to zero. In turn, with these large groups, peer relocations only start to happen at around delta 20, growing slowly, to then spike in the last delta value (i.e., delta 30). Considering that groups are big and peers enter the overlay as others leave, since there are practically no merges, relocations are no longer needed. Thus, peer relocations only start to happen in the last deltas values (that provide added freedom to peers).

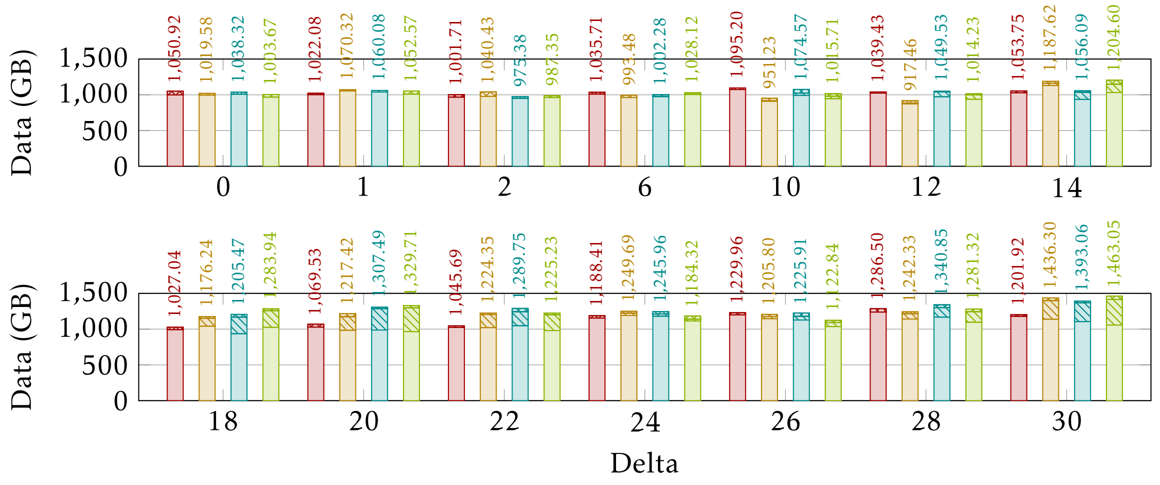
In Figure A.20, we can see the amount of data transfers resulting from merge, relocation, and group maintenance operations, for the *enter-exit* scenario. Until delta 16, the values are either zero or practically zero, thus we omitted them in the figure. This can be confirmed by the number of operations in the previous figure. Another relevant observation is that, since there is no need for merge operations, the vast majority of data transfers is due to peer relocations. On the one hand, this means that NPPR requires almost no data transfers in this scenario. On the other hand, this means that the peer relocations performed in this scenario are not actually necessary. They just happen due to the (excessive) freedom peers enjoy, given by the large deltas. As mentioned, like in Figure A.18, these plots also do not present the values for all the deltas. Similarly, we show a complete plot with all the values at the end of the appendix (see Figure A.28).



(a) 30% churn.



(b) 60% churn.



(c) 90% churn.

🟥 NPPR 🟨 Push 🟦 Pull 🟩 FPPR ☐ Merge ▨ Relocation ⊠ Maintenance

Figure A.18: Exit-only data transfers with group size XL (4–64) in Parsley.

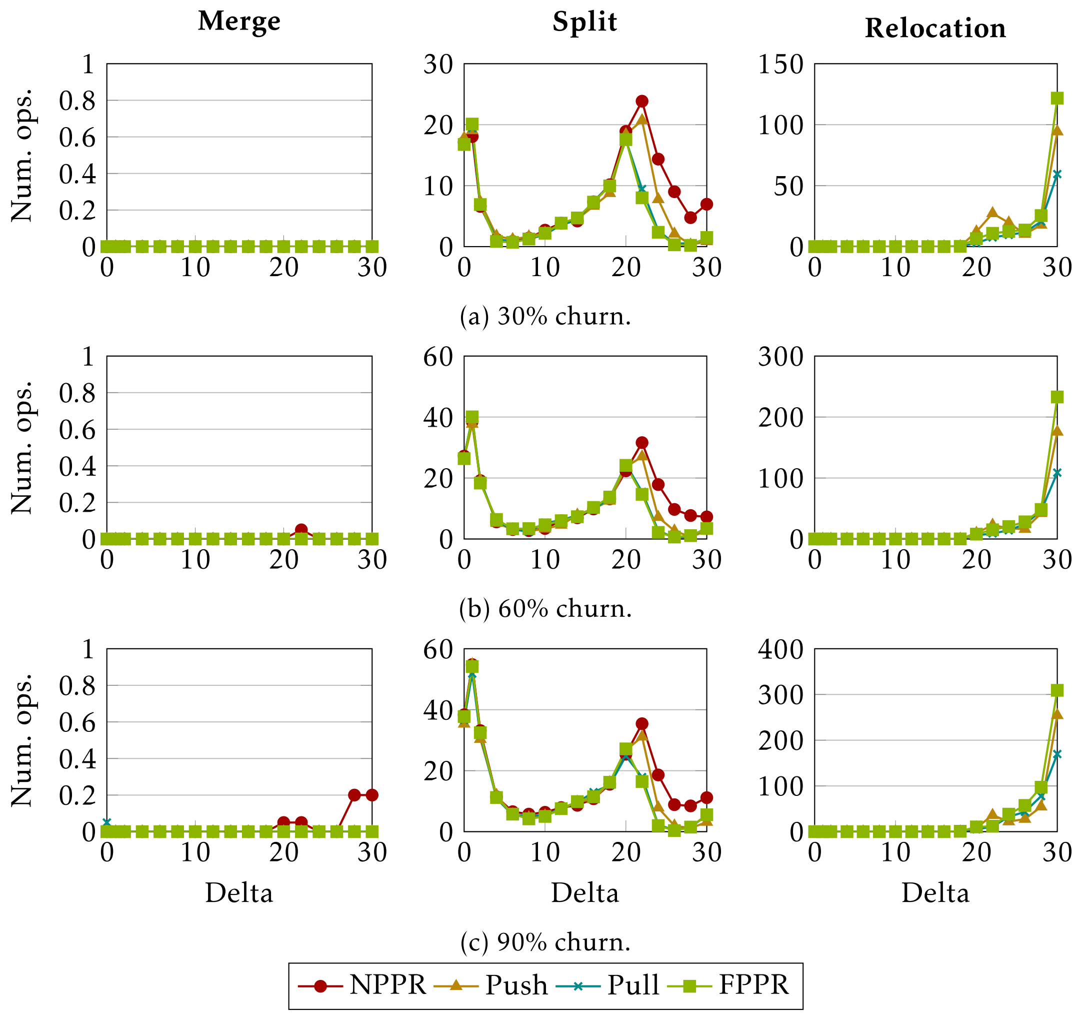
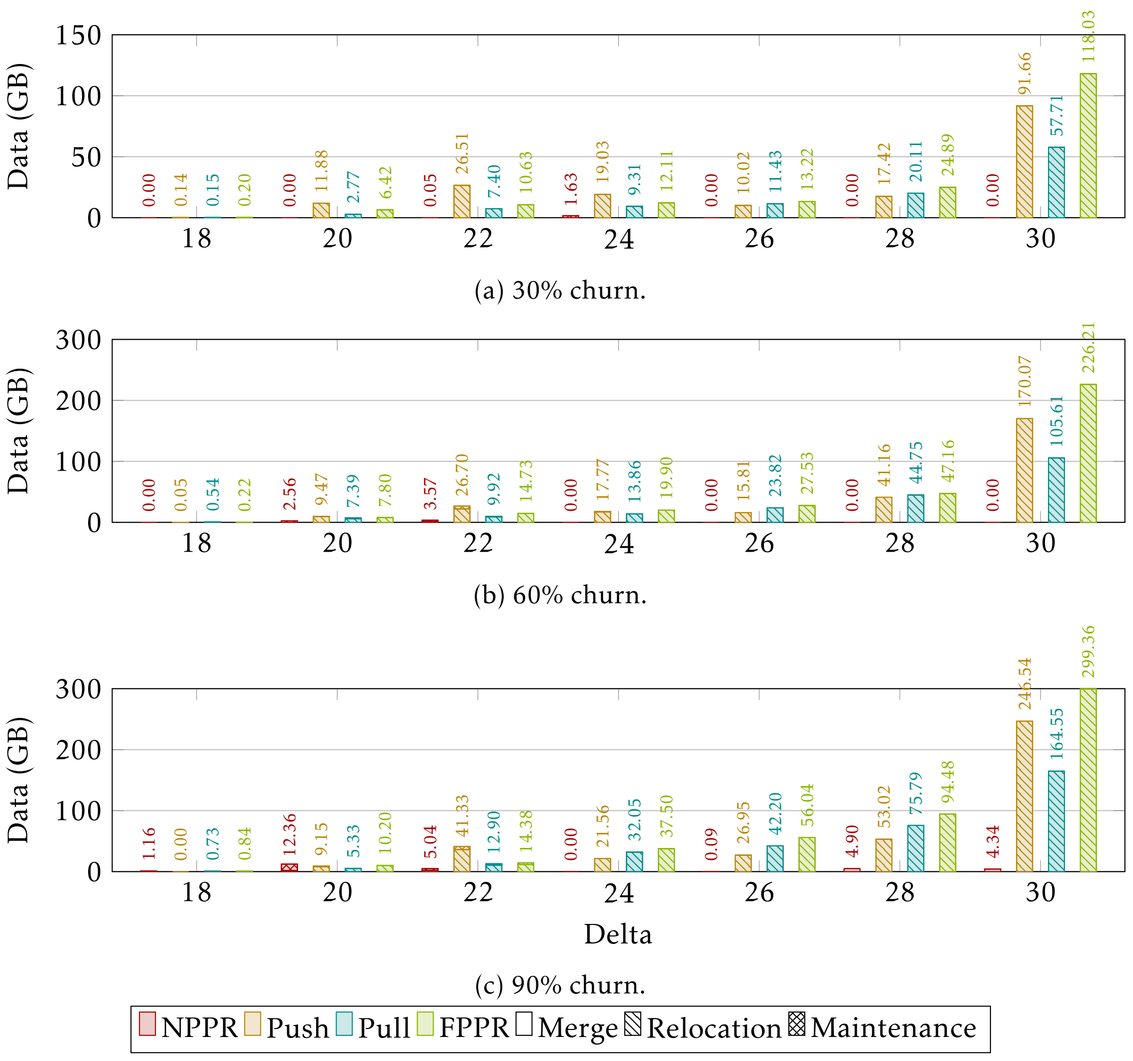


Figure A.19: Enter-exit topology operations with group size XL (4–64) in PARSLEY.

A.3 Concerning Big Groups

The bigger the group, the better it tolerates churn. That is, as seen in the many previous plots, bigger groups are more robust to churn, because its effects are less felt. The bigger the group, the more churn it can endure without requiring any type of action, such as topology changes. However, on the other hand, bigger groups do not come without issues. Naturally, since big groups mean a larger number of peers, they entail an increase in all group-related communication and data transfers. Also, bigger groups mean less overall groups in the overlay, since we keep the same number of peers. Specifically, big groups can encompass an increase in the following metrics:

- split-related traffic;
- maintenance traffic;
- per group state; and
- join state transfers.



(a) 30% churn.

(b) 60% churn.

(c) 90% churn.

Figure A.20: Enter-exit data transfers with group size XL (4–64) in Parsley.

To substantiate this claim, we present some plots next regarding these metrics. First, Figure A.21 shows the behavior of the split-related traffic as the maximum group size grows, for an example scenario (*enter-exit*, 90% churn, delta = 1). Comparing these values to the previously presented plots, it might seem negligible. Still, we can see the traffic related with split operations grows with the group size in a (supra-)linear way. To perform a split operation, it is necessary to notify all the peers in the group about the operation taking place, and they synchronize among them to speed up convergence (but without requiring data transfers). Thus, since groups are bigger, having a large number of peers, it ends up naturally requiring more communication.

Next, the values for the following plots across all configurations present a negligible difference. Thus, we report them as an average of all the configurations.

Figure A.22 depicts how the amount of maintenance traffic reacts as the maximum group size grows. Here, this traffic refers to intra-group maintenance, ring stabilization, fix fingers, and passive view maintenance (§2.4 and §5.3). However, mainly intra-group

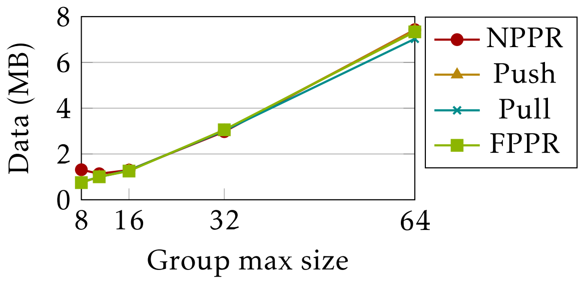


Figure A.21: Split-related traffic in Parsley (enter-exit, 90% churn, delta = 1).

and stabilization messages are influenced by the group size. In Figure A.22a, we can see the maintenance traffic for the *exit-only* scenario. Since there are only peers leaving the overlay, as time passes by, there are less peers and groups become smaller. Thus, it is natural that with high levels of churn, there is less maintenance traffic (since there are less peers). Nonetheless, this metric grows almost linearly with the group size. In turn, Figure A.22b shows the same metric for the *enter-exit* scenario. Since the overlay size is kept stable, with peers entering and leaving by the same amount, the behavior is the same for all the churn values (overlapping in the plot). In this scenario, the maintenance traffic also grows linearly wit the group size, reaching considerable values.

Figure A.23 presents the average amount of per group state, i.e., the average amount of data objects stored by group (in GB). Naturally, as already mentioned, larger groups result in less groups, since the number of peers is kept unchanged. Figure A.23a depicts this metric for the *exit-only* scenario. Here, we can see that the per group state grows almost linearly with the group size. Also, since no new peers enter the overlay, as the amount of churn increases, the number of peers per group decreases and so does the overall number of groups in the overlay. In the end, with high levels of churn, groups have to store more state. On the other hand, Figure A.23b shows this metric for the *enter-exit* scenario. Since the overlay size is kept stable, all the churn values behave identically (overlapping in the plot), with the per group state growing linearly with the group size. In both scenarios, the amount of per group state grows to considerable values accompanying the group size.

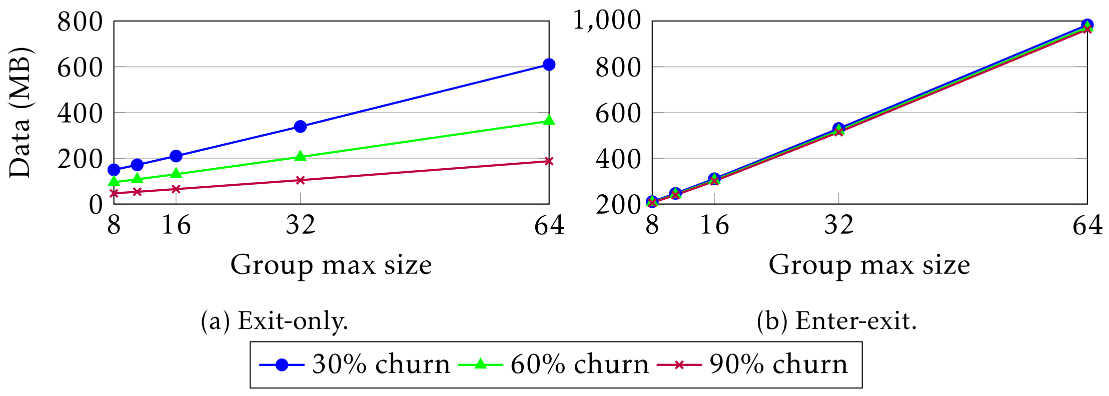


Figure A.22: Maintenance traffic in Parsley.

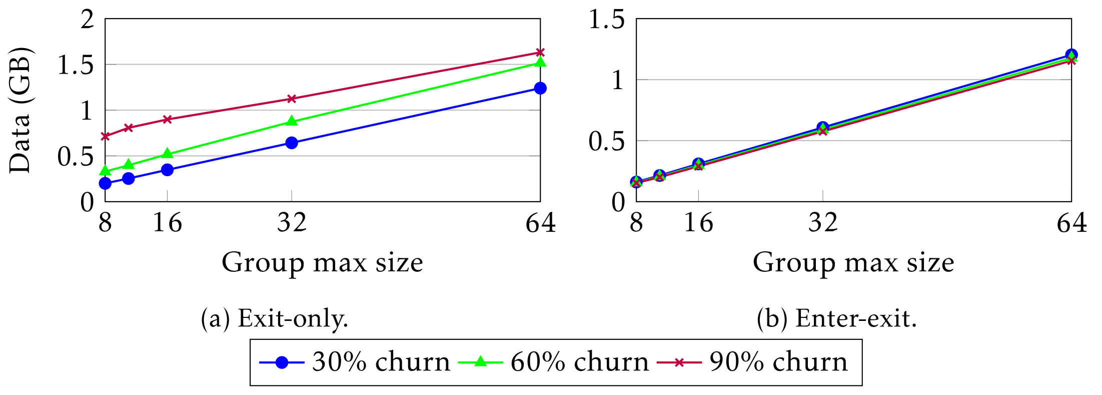


Figure A.23: Per group state in Parsley.

Lastly, in Figure A.24, we can see how the amount of data transfers due to the entry of peers in the overlay (what we call join state transfers) varies as group size increases. Naturally, this metric only applies in *enter-exit* scenarios. Looking into the figure, we can verify that the overall amount of data that needs to be transferred to peers joining the overlay grows linearly with the group size. For the same amount of peers, as groups become larger, there are less groups in the overlay and each group stores more data (as already seen in Figure A.23). In the end, when a new peer enters a group, it will need to synchronize with the peers already in the group to get itself up-to-date regarding all the state in the group (naturally including the stored data objects). Notice that the values presented in this plot represent massive amounts of data transfers due to joined peers—the scale in the y-axis is in gigabytes and is multiplied by 10 000. Thus, this metric should definitely be taken into account when choosing the group size ranges.

A.4 Discussion

First, regarding the delta size, we point out that a key aspect is to balance the amount of peer relocations with the decrease in merge operations. Otherwise, too many relocations can strip all the benefits from the peer relocation mechanism, and even start to become

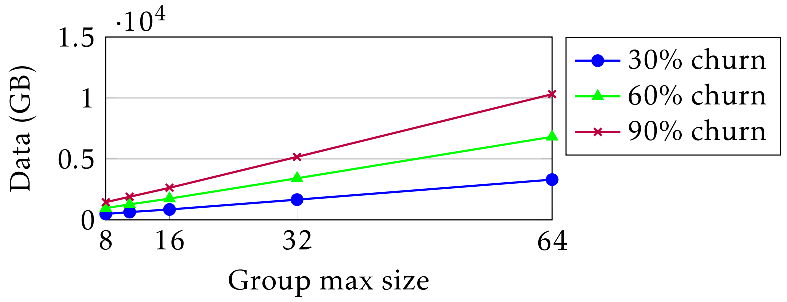


Figure A.24: Join state transfers in Parsley (enter-exit).

detrimental. Also, take into account that in some cases peer relocations grow linearly with the delta size. Here, we argue delta 1 shows the best trade-off between freedom to relocate and the decrease in the amount of merge operations. As mentioned throughout §A.2, this delta value exhibits the largest decrease in merge operations for the vast majority of the experimented scenarios, without awarding too much relocation freedom (so, without overwhelming the system with peer relocations, and being these useful relocations). Thus, showcasing that just a small degree of freedom is enough to significantly influence the number of required merge operations. Additionally, it also allows the largest average group size, making groups more churn-tolerant.

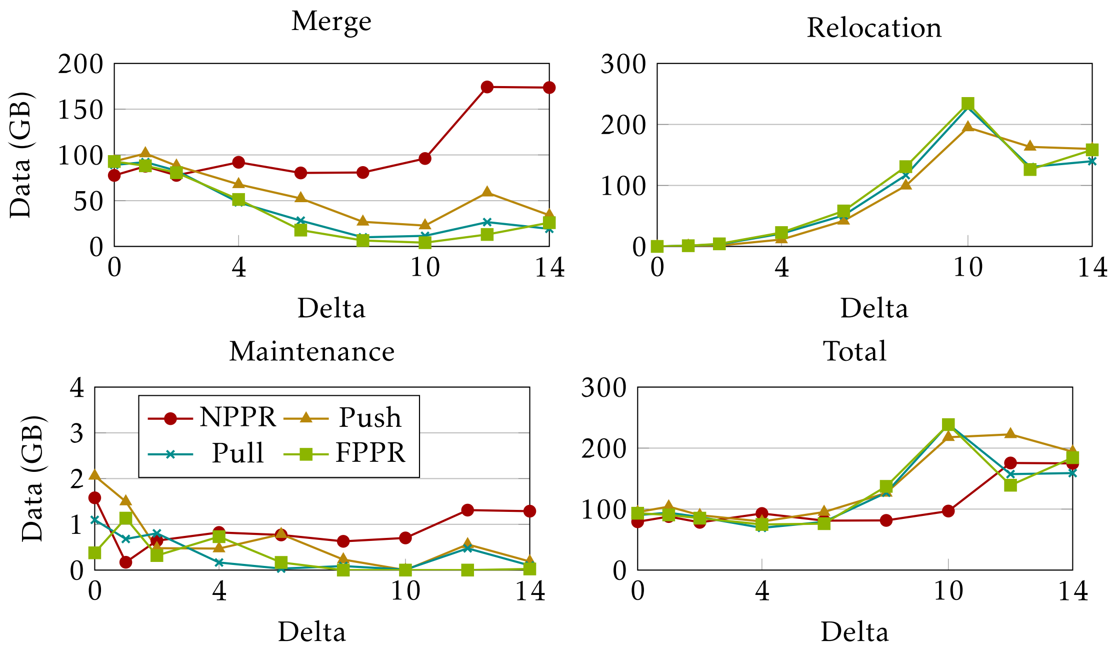
Then, concerning the group size, we argue for a range on the smaller side. Regarding the number of merges, size *extra large* and *large* are completely exaggerated, requiring almost no merges and demanding high costs in terms of group-related communication and data transfers (as shown in §A.3). In turn, the used scenarios are already churn-heavy, and the large sizes end up not needing much merges (sometimes not at all). Thus, to better showcase the benefits of our mechanisms for the used churn rates, we argue that size *small* (i.e., 4–11) is ideal. It presents a big enough average group size, requiring a decent amount of merges, without being too much. Also, we chose this range in order to give some balance between the two scenarios (*exit-only* and *enter-exit*), and across the various levels of churn.

In the end, with all these metrics in mind, the configuration parameters regarding the group size thresholds used in Parsley's evaluation reported in §5.5 are set to $l = 4$, $l' = 5$, $h' = 10$, and $h = 11$.

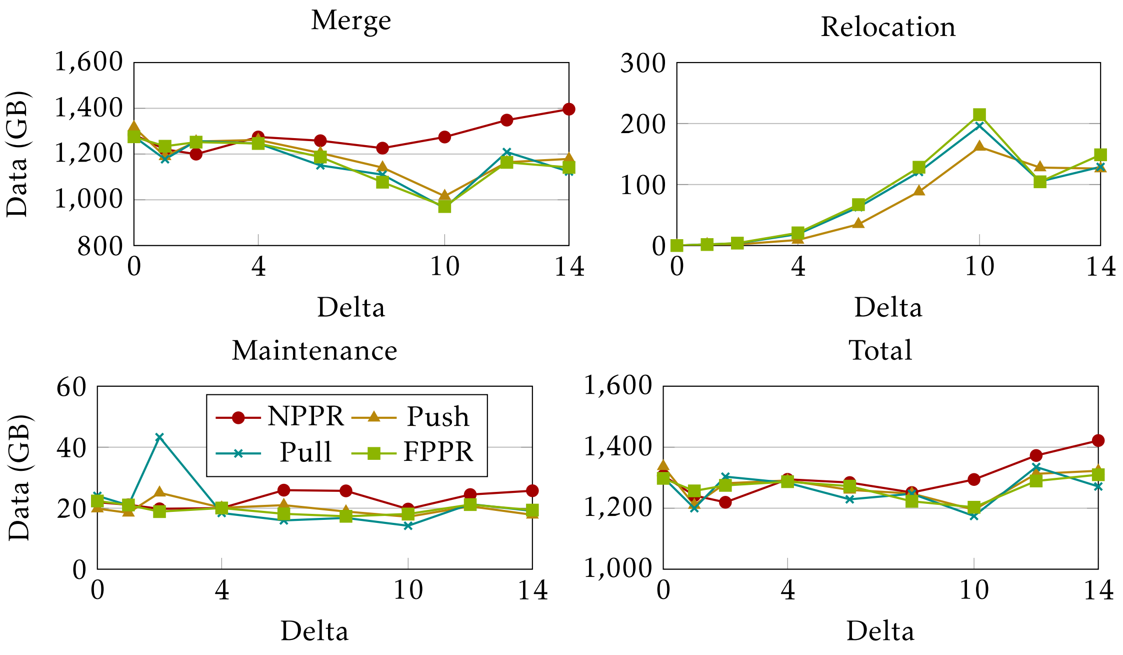
Other configuration parameters that can be experimented with are the periodic peer relocation check timer, and the relocation cool down period. Naturally, the smaller these are, the more freedom peers will have to relocate between groups.

A.5 Complete Plots

Here, we present the complete plots of some of the previously mentioned scenarios. Due to space and presentation concerns, the values regarding some deltas were omitted in those plots. Thus, for completeness sake, here we present them in its entirety (albeit in a different but more readable form).

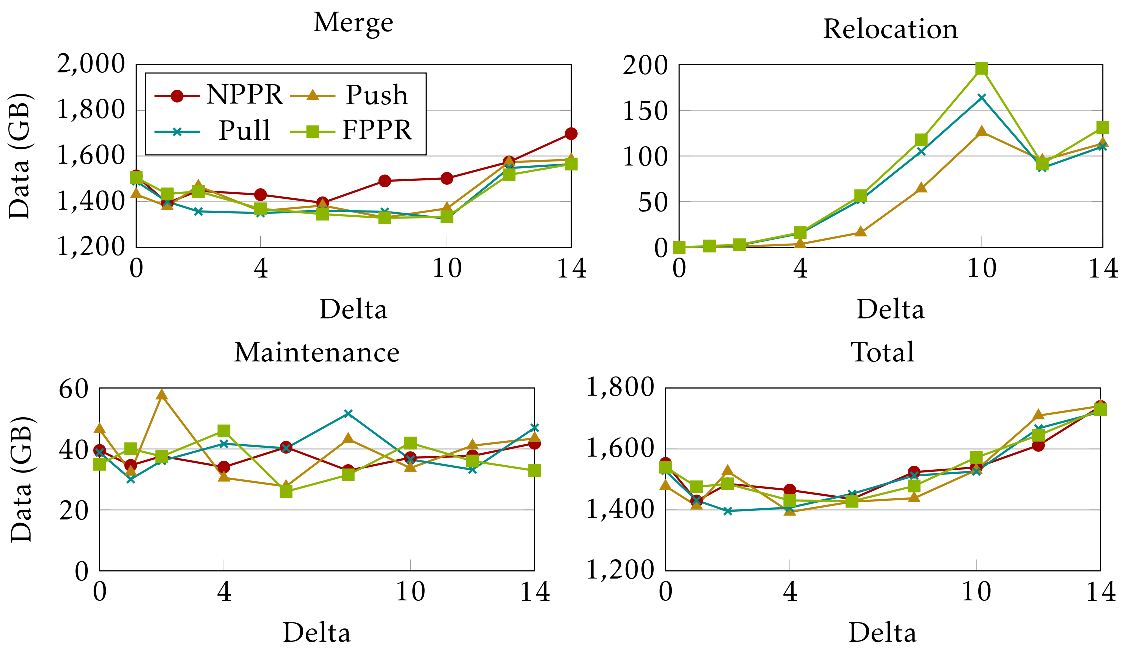


(a) 30% churn.



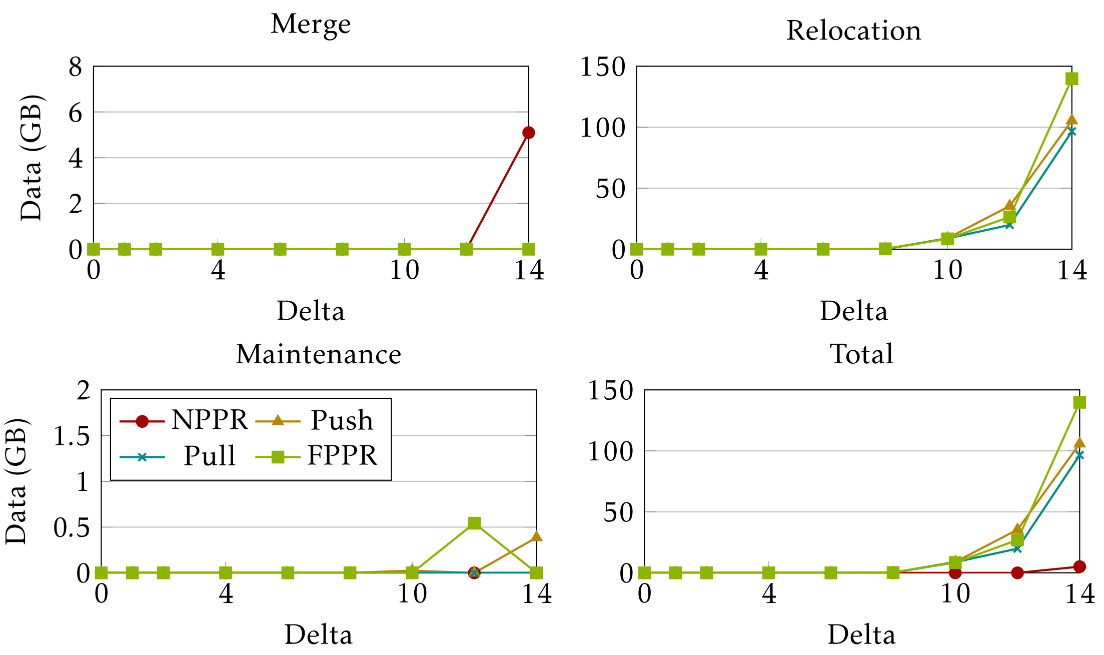
(b) 60% churn.

Figure A.25: Complete exit-only data transfers with group size L (4–32) in Parsley.



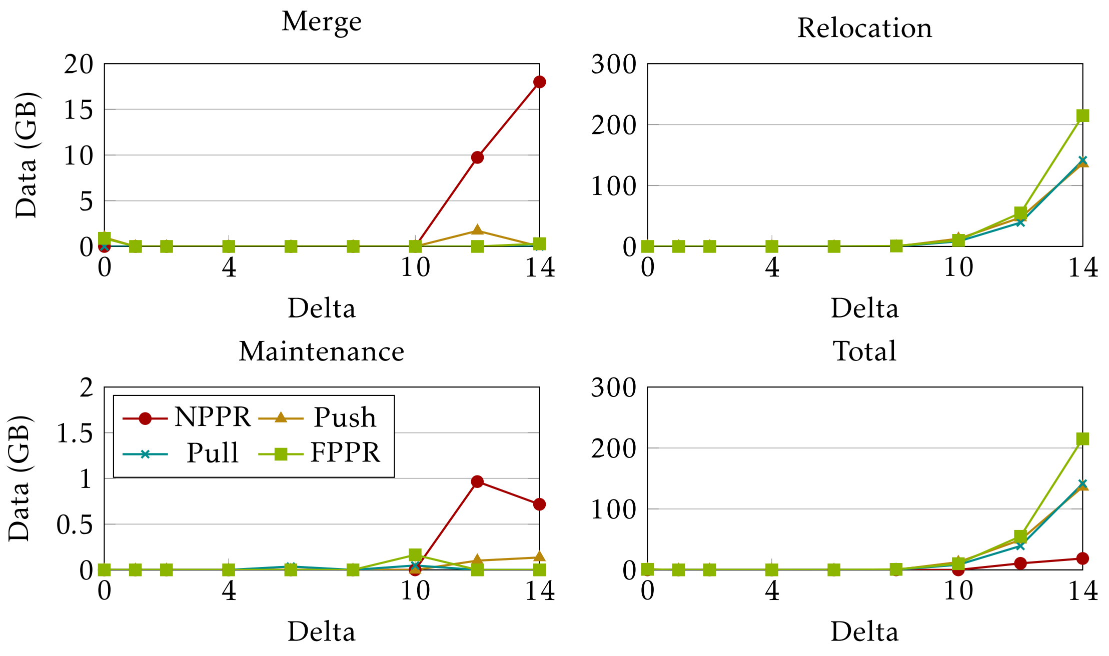
(c) 90% churn.

Figure A.25: Complete exit-only data transfers with group size L (4–32) in Parsley (cont.).

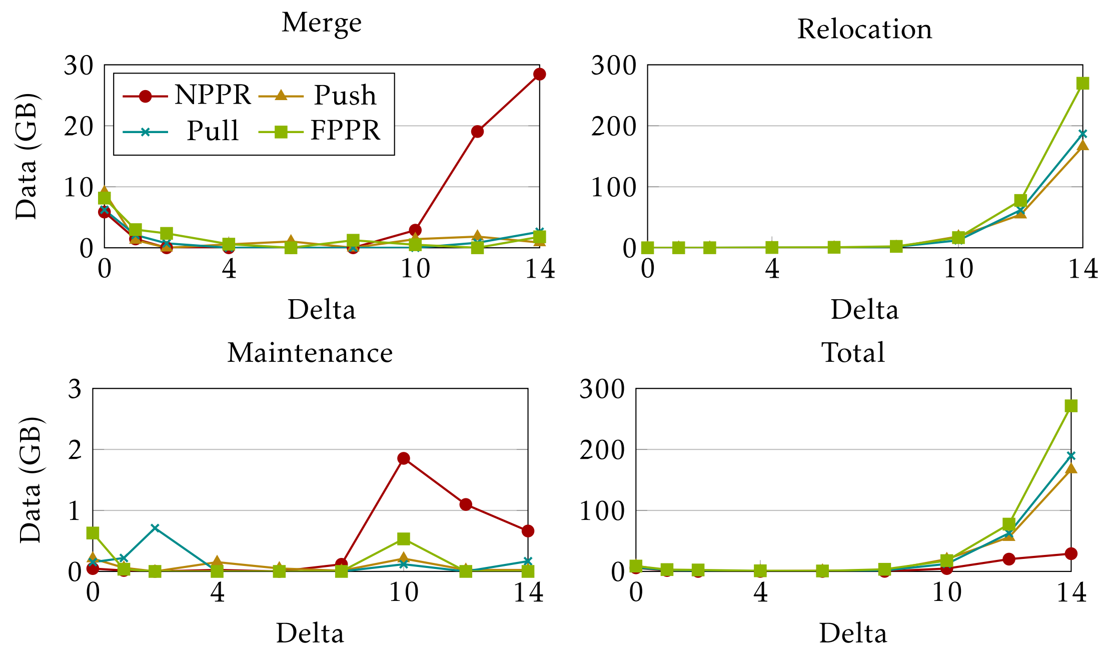


(a) 30% churn.

Figure A.26: Complete enter-exit data transfers with group size L (4–32) in Parsley.

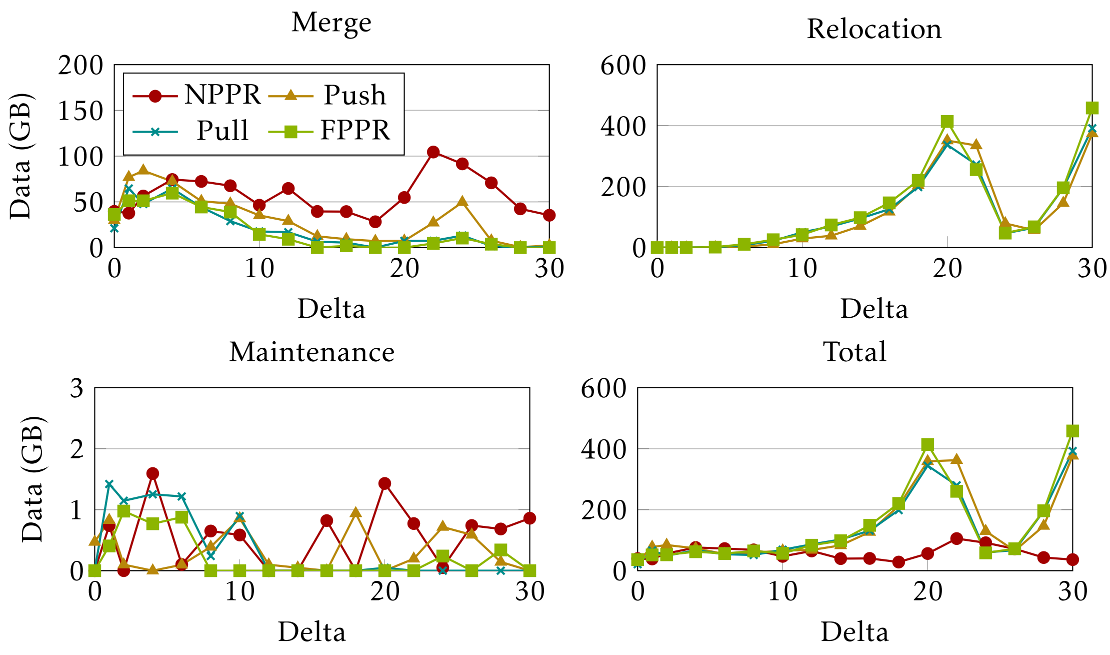


(b) 60% churn.

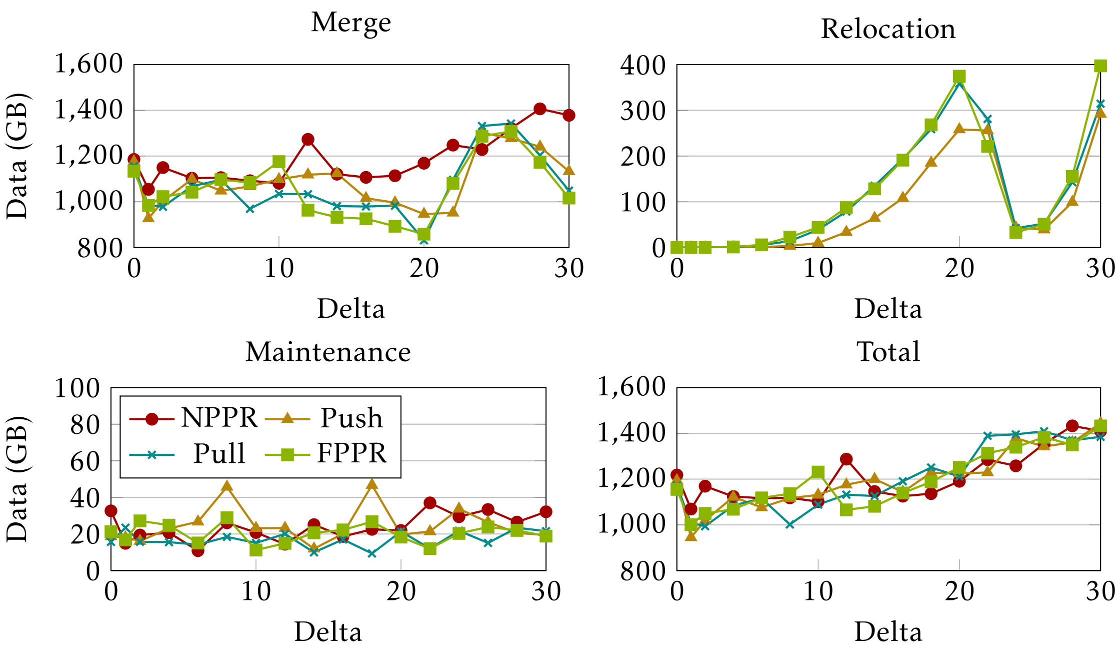


(c) 90% churn.

Figure A.26: Complete enter-exit data transfers with group size L (4–32) in PARSLEY (cont.).

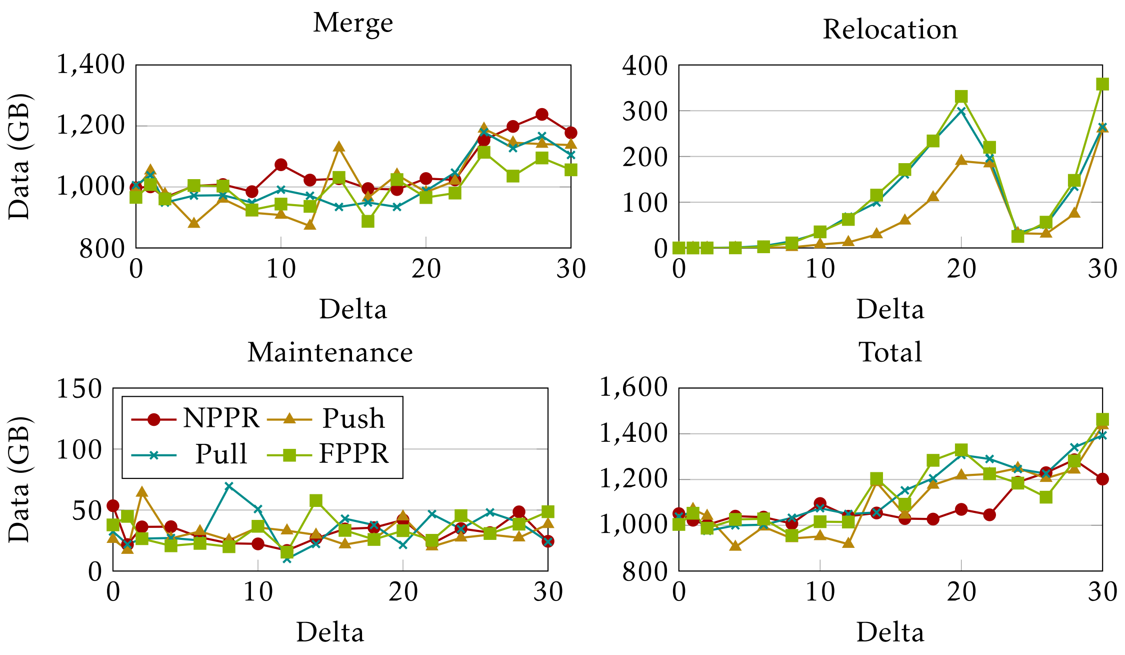


(a) 30% churn.



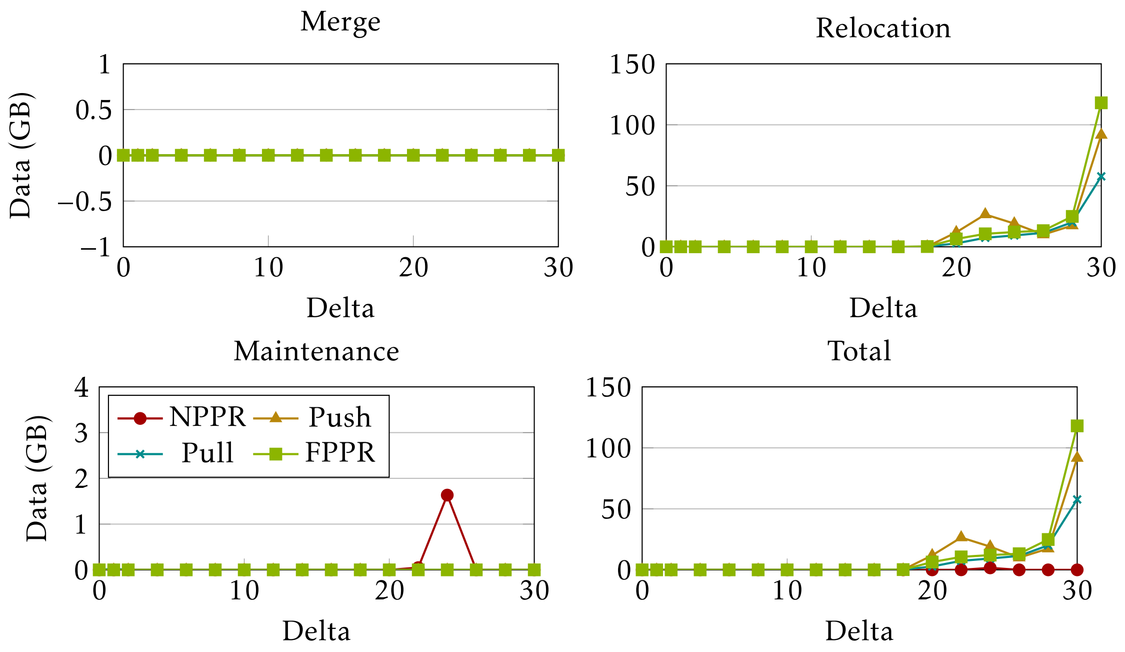
(b) 60% churn.

Figure A.27: Complete exit-only data transfers with group size XL (4–64) in Parsley.



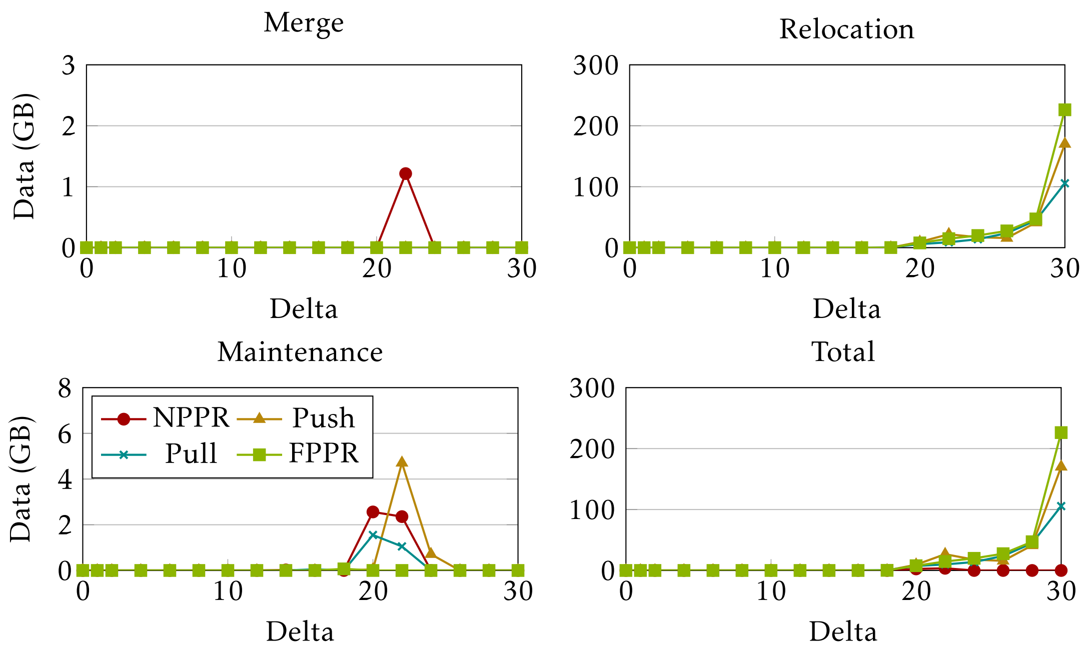
(c) 90% churn.

Figure A.27: Complete exit-only data transfers with group size XL (4–64) in Parsley (cont.).

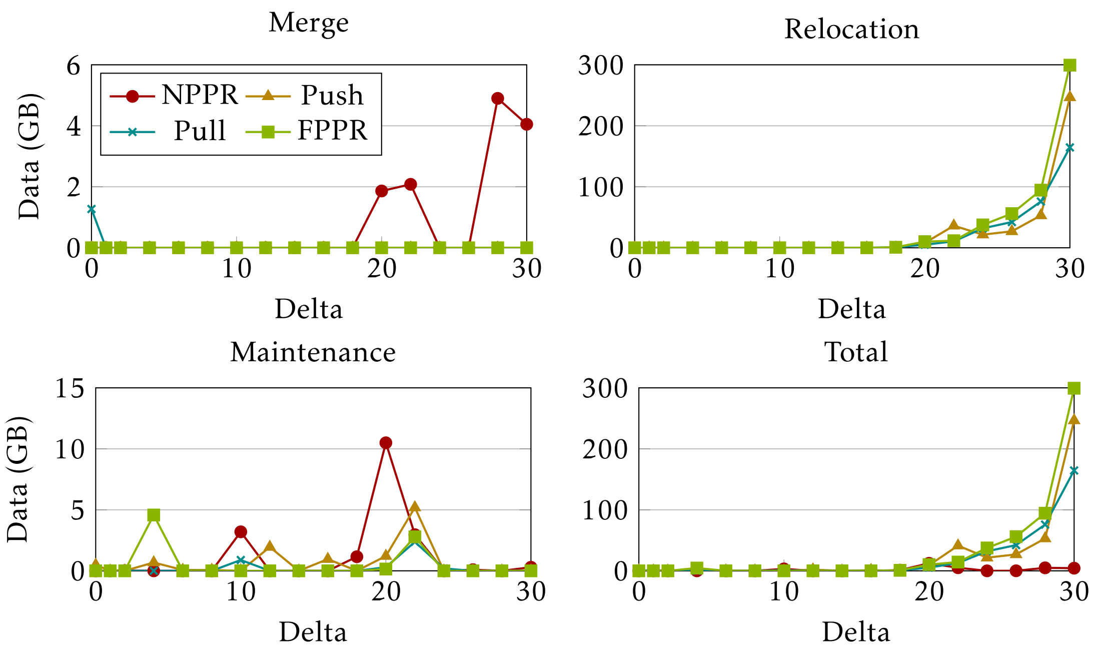


(a) 30% churn.

Figure A.28: Complete enter-exit data transfers with group size XL (4–64) in Parsley.



(b) 60% churn.



(c) 90% churn.

Figure A.28: Complete enter-exit data transfers with group size XL (4–64) in Parsley (cont.).