João Pedro Monteiro Morgado Dias

Bachelor in Computer Science

Adaptive Replica Selection in Mobile Edge Networks

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

Master of Science in
Computer Science and Engineering

Adviser: Hervé Paulino, Associate Professor, NOVA School of Science and Technology, Universidade NOVA de Lisboa

Examination Committee

Chair: João Moura Pires, Associate Professor, NOVA School of Science and Technology, Universidade NOVA de Lisboa

Members: Luís Veiga, Associate Professor, Instituto Superior Técnico, Universidade de Lisboa
Hervé Paulino, Associate Professor, NOVA School of Science and Technology, Universidade NOVA de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

November, 2020

Adaptive Replica Selection in Mobile Edge Networks

This document was created using the (pdf)LaTeX processor, based on the NOVAthesis template, developed at the Dep. Informática of FCT-NOVA by João M. Lourenço.

To my unconditionally supporting family.

Acknowledgements

My most profound gratitude goes towards p Prof. Hervé Paulino who has guided me throughout this journey, as well as prof. João Silva who has been as-present during the elaboration of the project.

Also, to the people who are closest to me and never failed to provide clarity during the time that I've been juggling my academic, professional and personal life.

This dissertation was written in the context of Project DeDuCe (PTDC/CCI-COM/32166/2017), financed by Fundação para a Ciência e Tecnologia.

Abstract

With the ongoing increase in mobile devices and the 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. Furthermore, having messages travel back and forth to a location that is physically distant from the user severely punishes applications with low latency or high bandwidth demands. 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.

As in a cloud environment, these applications are starting to make use of 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 (cached) in edge servers. The problem arises though, when there is no replica selection mechanism for data retrieval. The resulting herd behavior causes the computational load on the network to be poorly distributed, which combined with the unreliable wireless communication channels cause these systems to under-perform.

In thesis we propose Wasabi, an adaptive replica selection algorithm for MEC environments with the aim of decreasing *latency* and boosting both *throughput* and *energy efficiency* in MEC systems. Furthermore, we develop a whole replica selection framework to support Wasabi and its integration with Thyme GardenBed [14].

From our experimental results, we conclude that Wasabi performs better in dynamic environments than any of the presented baselines, including the cloud algorithm C3 [17] and its MEC variant, which make use of a similar set of metrics.

Keywords: Mobile Edge Computing, Replica Selection, Mobile-to-mobile, Mobile-to-edge

Resumo

Com o número de dispositivos móveis a crescer e as aplicações cada vez mais dependentes da cloud, estas infraestruturas têm-se tornado pontos centralizados de computação e armazenamento. Devido à quantidade de tráfego que é gerado - e recebido - nestas infraestruturas centralizadas, o congestionamento e atrasos na rede começam a torna-se evidentes. Além disso, a considerável distância física entre estas infraestruturas e os utilizadores inviabiliza algumas das aplicações com maiores exigências a nível de largura de banda e latência. Mobile Edge Computing (MEC) é um paradigma que procura resolver estas limitações trazendo os serviços cloud para mais próximo dos dispositivos móveis, de forma a reduzir a latência e poupar largura de banda no canal de comunicação principal.

Tal como nos ambientes cloud, estas aplicações começam a fazer uso de replicação para melhorar a sua qualidade de serviço. Como o conteúdo gerado pelos dispositivos móveis tem inicialmente um interesse localizado, os dados começam por ser replicados entre os dispositivos móveis e só mais tarde, quando se começam a tornar populares, é que são eventualmente replicados (cached) em servidores edge. No entanto, o problema surge quando não existe qualquer tipo de mecânismo de seleção de réplicas para descarregar esses dados. Pela ausência de tal mecânismo, é costume observar-se um mau balanceamento de carga entre as réplicas disponíveis, o que combinado com canais de comunicação instáveis, degrada a performance destes sistemas.

Com esta tese nós propomos Wasabi, um algoritmo de seleção de réplicas adaptativo no âmbito de sistemas MEC, com o objetivo de diminuir latências e melhorar ambos o *throughput* e a eficiência energética destes sistemas. Dos nossos desenvolvimentos resulta também uma *framework* para desenvolver mecânismos de seleção de réplicas e sobre a qual construímos a integração do Wasabi com o sistema Thyme GardenBed [14].

Através dos nossos resultados experimentais fomos capazes de concluir que o nosso algoritmo faz melhores seleções em ambientes dinâmicos que qualquer outra *baseline* definida para efeitos de comparação, incluíndo o algoritmo cloud C3 [17] e a sua variante MEC, e que usam um conjunto de réplicas semelhante às do Wasabi.

Palavras-chave: Mobile Edge Computing, Seleção de Réplicas, Mobile-to-mobile, Mobile-to-edge

Contents

List of Figures

List of Listings

Acronyms

AP Access Point

API Application Programming Interface

AR Augmented Reality

DHT Distributed Hash Table

EWMA Exponentially Weighted Moving Average

IoT Internet of Things

JVM Java Virtual Machine

MEC Mobile Edge Computing

TDLS Tunneled Direct Link Setup

TTL Time to Live

UUID Universally Unique Identifier

VR Virtual Reality

Chapter 1

Introduction

We start this work with its context and motivation. We then move to introducing the Mobile Edge Computing (MEC) [1] paradigm, followed by a brief description of the problem and its manifestation in a particular publish-subscribe (PS) system. We then discuss our proposed solution before ending the chapter with a structure overview for the remainder of the document.

1.1 Context and Motivation

In recent years, users have been increasingly adopting smartphones as their primary internet-enabled device. In 2016, analysts disclosed the first report stating that mobile devices surpassed desktop in terms of internet usage worldwide [15]. Also, the advent of IoT and wearable devices has been contributing to this trend.

Nowadays, many mobile applications rely on services hosted in the cloud. In most cases, all data traffic is routed through the core network to a base station which delivers the content to mobile devices. Even though smartphones' hardware capabilities have been greatly increasing year after year, mobile communications still remain a bottleneck for most applications.

All the network requests generating from these internet-enabled devices, as well as desktop computers, are being processed by the same cloud infrastructure in a centralized fashion. And even though geo-replication mitigates this scalability issue, the number of concurrent network requests is so big that they are still competing for the servers' processing power and channel throughput.

This reliance on a cloud data center is specially not feasible for applications that require end-to-end delays to be tightly controlled. This is the case for the newer emerging types of applications with high bandwidth demands (such as AR and VR).

Additionally, wireless communication technologies used in mobile environments - such as Wi-Fi, Bluetooth and 3G - are unreliable, slow and congestion-prone by nature when compared to the wired medium counterpart [21].

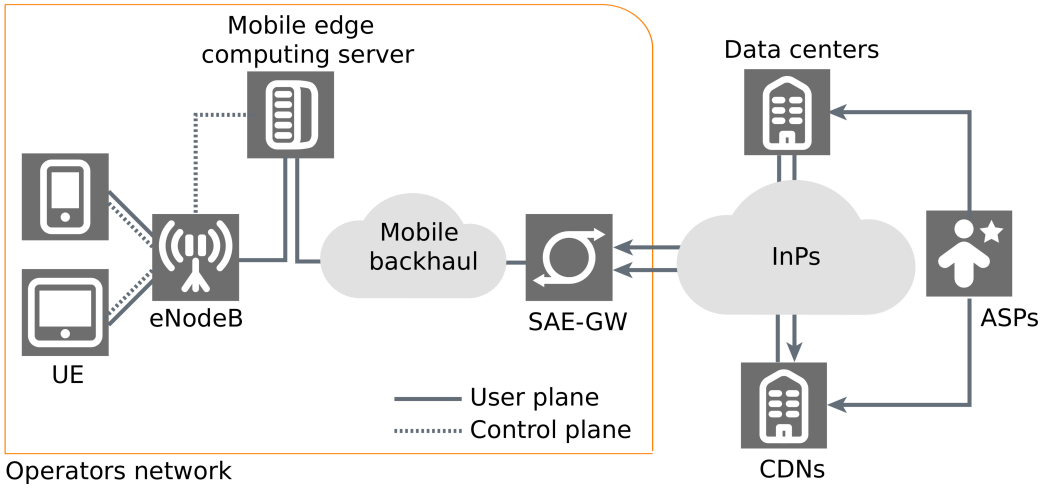


Figure 1.1: Mobile Edge Computing Topology: 1) Mobile end users using User Equipment (UE), 2) Network operators owning, managing, and operating base stations, MEC servers, and the mobile core network, 3) Internet infrastructure providers (InPs) maintaining Internet routers, 4) Application service providers (ASPs) hosting applications within data centers and content delivery networks (CDN).

All the previous have been the catalyst to the introduction of Edge Network. In the context of this work, when referring to the Edge, we'll be talking about Mobile Edge Computing (MEC).

1.2 Mobile Edge Computing

MEC brings cloud services closer to the mobile clients, i.e. the edge of the network, by leveraging on the storage and processing power of smaller computational servers that are deployed in the base stations of cellular networks. By being closer to the data sources, MEC servers effectively reduce end-to-end delays and save backbone bandwidth for those cases that strictly need to reach the main infrastructure. This paradigm allows for the removal of network bottlenecks and the support of previously mentioned emerging applications with strict end-to-end delay requirements. Furthermore, other iterations are moving away from single node access points towards using the client's mobile devices as actual Edge nodes, harnessing their processing, storage and routing/communication capabilities, in order to create a complete local system [21]. Fig. 1 depicts the MEC ecosystem and the integration of MEC servers into the mobile network topology.

Another interesting point for our work is that by being placed at the mobile edge, MEC servers are capable of collecting real-time network data like cell congestion, subscriber locations and movement directions [2].

Regarding the applications that take advantage of the MEC, these can fit into one or more of the application classes in Fig. 2. We will be focusing on *Edge Content Delivery*, using metrics like *Power Consumption*, *Delay* and *Bandwidth* to evaluate each solution.

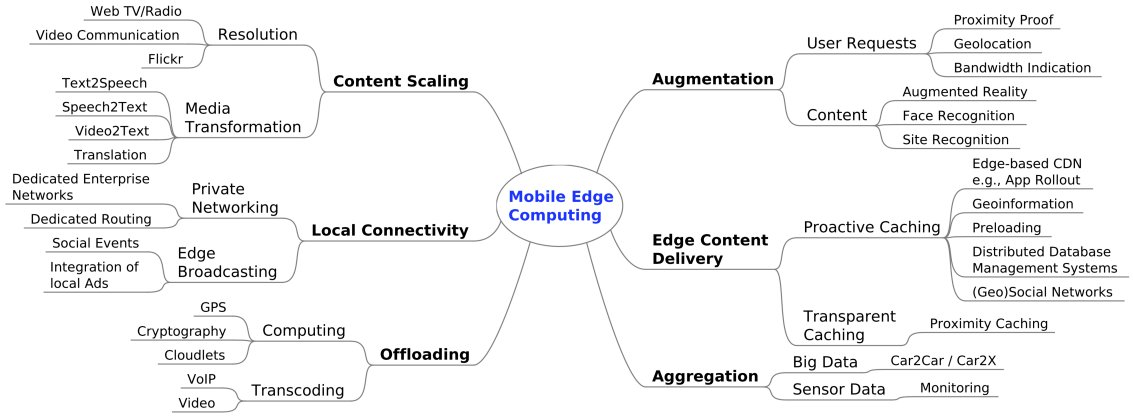


Figure 1.2: Mobile Edge Computing: Application Classes

In spite of both processing and battery capacity increases that we've been seeing in mobile devices over the years, for most use cases it is still considered a good practice to *offload* resource- or power-intensive tasks to remote services. We should delegate computation to external services when we either can't execute those computation in time with our local resources or such practice will noticeably favour battery life. If no MEC server is available, mobile devices can downgrade to a more distant MEC server, the main servers or fallback to local computations. Also, mobile applications have to be aware of the fact that MEC servers are deployed in a centralized way and, since the mobile user might move from its current geographical position, connectivity between MEC servers and mobile device is constrained. Thus, applications that rely on MEC services have to be mobility-aware [2].

Edge Content Delivery refers to the fact that MEC servers can operate as local content delivery nodes and serve cached content. This caching mechanism can be *Transparent*, that is, neither the mobile device nor the service are aware it; *Proactive*, meaning that the service provider will have some content already cached on their MEC servers under the expectation that such content would otherwise create a heavy load on the network; and *Preload*, which is the act of caching data before it is (potentially) requested by the end user. In contrast to proactive caching, it is the end users' actions that determine whether (and which) content gets upper handedly cached in the MEC servers. In the context of Mobile Edge Computing, pre-loading is shifted from mobile devices to MEC servers in order to decrease power consumption.

To sum up, the introduction of MEC servers in the initial network scheme is motivated by end users' benefits from reduced communication delays. Due to their proximity to end users, MEC servers allow for new kinds of applications to be considered. Also, service providers can now host their services at the edge, effectively scaling their services and saving backbone bandwidth. It also enables the integration of additional, congestion- or user-related information into the traffic flow.

1.3 Problem

Because in a typical MEC system nodes don't dispose of many hardware resources and communication channels are somewhat unreliable compared to a cloud environment, slowdown effects due to congestion or computational load might still be felt. Furthermore, metrics like power efficiency have to be taken into account when dealing with mobile devices. To meet applications' low-bandwidth demands, intelligent resource management processes have to be in place. This is specially true for MEC applications that employ replication to enhance their quality of service by offering better availability, scalability and reliability. This, however, can do more harm than good if there's no proper replication and replica selection strategy to store and retrieve data. In the scope of this thesis, we focus on MEC systems already employing some kind of replication technique to answer the following question: "Where should I retrieve the data from?".

Some edge-enabled peer-to-peer architectures that allow the creation of collaborative storage systems in bound-restricted geographical areas have recently emerged. Thyme GardenBed [14] is such an example. Thyme presents a novel time-aware approach to Publish/Subscribe systems for inherently mobile devices where the publish and subscription brokers are the clients themselves. Thyme relies entirely on P2P communication means, which are limited in range. Gardenbed, an edge infrastructure service that combines device-to-device and device-to-edge interactions, with the goal of optimizing access to popular data and allow such data to be available not only to users inside the same *local area* but also the same *local region*. Different local areas can access each others' popular data by having their infrastructure nodes (MEC servers) connected. Whenever a mobile clients wants to retrieve a popular item, it will always try to do so from the infrastructure.

The combination of device-to-device and device-to-edge interactions presents alternative channels for data retrieval. How can we leverage such channels to avoid bottlenecks? Can we fallback from requesting content from the infrastructure to requesting it from peer devices? When should we do it? And what if the problem doesn't lie within the MEC server but rather in the *local area*'s access point? Can we drop Wi-Fi communications altogether?

1.4 Solution

Considering the problem presented in the previous section, we think such systems can benefit from a service that uses network and computational-load metrics in order to predict the best replica(s) to contact. Thus, our aim was to implement a system-agnostic decision mechanism to determine the best data-fetching strategy in order to avoid oversaturated channels and preserve battery capacity whenever possible.

With this thesis we propose a low-profile replica selection framework consisting of the following high-level components:

1. A server-side metrics aggregation component that collects system metrics on demand;

2. A metric collector component that is programmatically configured to collect a metric value on demand and can be hooked into the aggregator;

3. A client-side replica classifier component that consumes metrics and is able to sort a set of nodes from most to least reliable according to the configured scoring logic;

4. A client-side metric observer component that can reactively compute metric values from system events.

We then propose an algorithm that takes predefined network, resource-usage and device-specific metrics to predict with a high degree of certainty which is the best replica to contact.

And finally, we use Thyme GardenBed [14] as an integrating system to evaluate our solution.

1.5 Contributions

Throughout the development of this thesis we made the following contributions:

1. A low-profile replica selection framework that is system agnostic;

2. The integration of this framework within Thyme GardenBed;

3. Test results obtained through simulation that compare latencies, energy efficiency and throughput from the current Thyme GardenBed implementation to the one with our module integrated an using different baselines/algorithms.

1.6 Document Structure

The remainder of this document is structured as follows:

- Chapter 2 - Related Work: Here we start by analysing replica selection algorithms in general. We focus on algorithms that are currently being used in practice, classifying them on the metrics they use and how that information is available to the client. Then we look at MEC systems, how they replicate and their replica selection process. And finally we look at different device-to-device communication protocols.

- Chapter 3 - Thyme: In this chapter we look at Thyme, GardenBed and finally its joint implementation, Thyme GardenBed [14].

- Chapter 4 - Proposed Solution: This chapter goes in detail into the architecture of our replica selection framework and how its components can be composed to easily

and flexibly bring replica selection capabilities to any application. We also detail how we have integrated it into Thyme GardenBed [14].

- Chapter 5 - Evaluation: In this chapter we show some test results and explain how we've evaluated our module integration with Thyme GardenBed [14].
- Chapter 6 - Conclusions: And finally we close with some final thoughts and suggestions for further work and investigation.

Chapter 2

Related Work

In this chapter we will be looking at already developed work regarding **dynamic replica selection**. We focus on replica selection algorithms towards distributed key-value stores for cloud computing because, despite MEC environments being more volatile in comparison to the cloud, their replication models fundamentally impose similar challenges. We also look at other scenarios such as grid computing, and understand how general these algorithms can be. Following the approach of [3], we start by summarizing current replica selection algorithms and classifying them into three categories: information-agnostic, client-independence and feedback.

Then, we take a look at some MEC systems to understand how they replicate data and how they choose replicas to retrieve data.

Finally, we do a brief comparison of existing direct-channel communication protocols for mobile-to-device interactions.

2.1 Dynamic Replica Selection

2.1.1 Overview

Nowadays, data is usually replicated and distributed across servers for parallel access and scalability. Thus, several replica servers might be available for the service. Two important challenges in data replication techniques are: (i) replica placement and (ii) replica selection. Replica placement is the problem of placing duplicate copies of data in the most appropriate node; Replica selection is the problem of selecting the best replica site for users to access the required data during execution. Although both play a big role in the overall system performance, the focus of our work is on replica selection.

In a key-value store environment, the value of each key is typically replicated and distributed across a group of replica servers. A client can select any one of these replica servers for each key-value access. On the other side, servers receive keys from different clients. When the server is busy, the newcome keys will be put into the waiting queue. After a key is served, the corresponding value will be returned back to the client. This

pattern is very identical for other scenarios which makes it easy to extrapolate to those. To avoid these waiting queues, the client must be able to select the proper replica server.

We will now classify the most common algorithms and replica selection mechanisms in the context of key-value stores:

Information-Agnostic: These are algorithms that pick a replica in an uninformed way, not taking into account any extra information or external metrics. Examples of such algorithms are Fixed, Random and Round Robin. Fixed always targets the same replica, only taking into account any of the others if the fixed one is unavailable; Random, as the name suggests, randomly selects a replica from the available pool and finally Round Robin continuously iterates through the replica set, picking a different one for each access. Although Round Robin is more load conscious from the client's perspective, neither of these algorithms take into account any external information or measured metrics, which might results in recurrent bad decisions specially for volatile scenarios like the ones we target.

Client-Independence: In these category, algorithms take into account metrics independently measured by the client, without any aid from the servers. According to the authors of [3] and further studied related work [17] [16], some classic helpful metrics are:

- The round trip time (RTT) of network;
- The response time (RPT) of each key-value access, which involves not only the RTT of network but also the service time and the waiting time at server;
- The outstanding keys (OSK), which has been sent out to replica server but the corresponding value has not been received.

We can use any of these metrics directly to choose an appropriate replica or compute some probabilistic measure with the combination of various metrics. For example, Riak [10] algorithm lets the client choose the server with the least number of OSK. There's also the two choices way [7], where client chooses two replica servers randomly at first, and then chooses one of these two replica servers according to above information. MongoDB has a similar approach, where it selects the nearest replica servers by RTT at first, and then randomly chooses one of them [8]. One downside for this category is that the available information such as RTT and RPT may not be fresh at the time of picking a replica. This can happen simply to the lack of interactions of the client with the system within a certain time frame, which will yield no metrics for that period. This is also something that is brought to our attention by the authors of C3 [17] and will be further discussed in the bellow section dedicated to this state-of-the-art [16] algorithm.

- **Feedback:** This category builds on top of the previous one by adding *pigggybacked* information with the returned values from the server, which means that both clients

and servers form a feedback system. A great example is C3, which observes that the fastest replica server is not only determined by its load or the observed latency, but also depends on the performance of that replica server, and thus piggybacks the service time of each key to adapt with the time-varying performance of replica servers. We will be discussing C3 more in-depth in the following section. The authors of [3] also propose L2, which yields similar performance compared to C3 in simulation results, but is much simpler. We will also be discussing this algorithm in sequence with C3.

Finally, we would like to mention some additional methods to reduce data-access latency, such as request duplication and reissue. These can be effective in some cases but are overall more hurtful when there's no proper replica selection algorithm in place. Reissuing requests but selecting poorly-performing nodes to process them increases system utilization in exchange for limited benefits and is the cause for more herd-behavior. We argue that these techniques can be effective as a complement to replica selection algorithms.

The following sections further describe the most relevant algorithms and mechanisms for dynamic replica selection.

2.1.2 C3

C3 [17] is an adaptive replica selection mechanism that is robust in the face of fluctuations in system performance. Because servers exhibit performance fluctuations over time, replica selection needs to quickly adapt to changing system dynamics. This is where most replica selection algorithms (e.g. Cassadra's Dynamic Snitching strategy, which computes replica scores at fixed discrete intervals) fail to deliver: they do not implement a reactive solution. Thus, C3 combines two mechanisms in order to carefully manage tail latencies in a distributed system: (i) a load-balancing, replica ranking scheme that is informed by a continuous stream of in-band feedback about a server's load, and (ii) distributed rate-control and backpressure.

With replica ranking, clients individually rank servers according to a scoring function, with the scores serving as a proxy for the latency to expect from the corresponding server. Servers piggyback information about their queue size and approximate service time on each response to a client, and clients maintain a weighted moving average of these metrics. There's also a *concurrency compensation* that is calculated to account for both the existence of other clients in the system and the number of requests that are potentially in flight. If *concurrency compensation* is not taken into account for the estimation of each server's queue-size, replica selection gets prone to herd behaviors. The number of requests that a client has pending over a given server also weights on the server's score. It was also decided to penalize scores over queue sizes using a non-linear function. This is because for a given server A with a service time n times faster than server B, such server would be able to get the same score as server B while holding a queue n times longer if we were

to use a linear function such as multiplying the two values. If the service time of A then increases due to an unpredictable event such as a garbage collection pause, all requests in its queue would incur higher waiting times.

Because replica selection alone cannot ensure that the combined demands of all clients on a single server remain within that server's capacity, clients rate-limit requests to individual servers. If the rates of all candidate servers for a request are saturated, clients retain the request in a backlog queue until a server is within its rate limit again. Every client maintains a rate-limiter for each server, which limits the number of requests sent to a server within a specified time window. Such limit is called *sending rate*. They also track the number of responses being received from a server in an interval of the same length (*receive rate*) and then the rate-adaptation algorithms tries to match both rates. Upon receiving a response from a server s, the client compares the current sending and receive rates for s. If the client's sending rate is lower that the receive rate, it increases its rate according to a cubic function. At any time, if the algorithm perceives itself to be exceeding the server's capacity, it will reduce its sending rate. Lastly, given that multiple clients may potentially be adjusting their rates simultaneously, the step sizes of the rate increase is capped.

The C3 replica selection process is as follow: When a request is issued at a client, it is directed to a replica selection scheduler. The scheduler uses the scoring function to order the subset of servers that can handle the request, that is, the replica group. It then iterates through the list of replicas and selects the first server s that is within the rate as defined by the local rate limiter for s. If all replicas have exceeded their rate limits, the request is enqueued into a backlog queue. The scheduler then waits until at least one replica is within its rate before repeating the procedure. When a response for a request arrives, the client records the feedback metrics from the server and adjusts its sending rate for that server.

We consider C3 to be a good base approach for our problem because it can easily be adapted to MEC environments and does not introduce any extra messages in the network nor requires heavy computations that would otherwise clog the nodes and system progression.

2.1.3 L2

Based on the insights obtained from their performance analysis, the authors of [3] propose a new algorithm: L2. From their simulations, the authors concluded that:

- RPT is useful for the selection of fastest replica server, but may lead to the herd behaviors;

- OSK is helpful to both the selection of the fastest replica server and the avoidance of the herd behaviors to some extent.

Similar to Riak's algorithm, L2 first selects the replica servers with the least number of OSK, and then sorts this subset of replica servers according to their response times. From the resulting set, the replica with the smallest RPT and the least number of OSK, in total, will be picked. In this way, L2 gives consideration to both the selecting of the fastest replica server and the load balance among replica servers.

L2 is simpler than C3, as it doesn't need any feedback information or the rate control mechanism. However, L2 can achieve a similar best performance in terms of tail latency like C3. The authors conclude that the complicate rate control mechanism of C3 itself is not helpful to reduce the tail latency. Moreover, L2 can also avoid the herd behaviors like C3, with the help of the OSK. Therefore, L2 might also be an interesting baseline for our solution.

2.1.4 NetRS

The conventional scheme we've seen so far is each client being a Replica Selection Node (RSNode). A RSNode independently selects replicas for requests based on its local information, including the data collected by itself (e.g. the number of pending requests) and/or the server status in responses. This approach, however, has some pitfalls:

- Considering that one client typically sees a small portion of the traffic, they are likely to select a poorly-performing server for a request due to its inaccurate estimation of server status;
- Servers may suffer from load oscillations due to "herd behavior"(multiple RSNodes simultaneously choosing replica), which is positively correlated to the number of independent RSNodes.

NetRS [16] is a framework that enables in-network replica selection for key-value stores in data centers. To overcome such pitfalls, NetRS offloads tasks of replica selection to programmable network devices. Because these devices are much fewer than the end hosts, they tend to process more messages and thus have fresher and more abundant network information. This also minimizes the "herd behavior"since now one single device can select replicas for multiple clients. There's also attention in minimizing the number of RSNodes.

There are two main components:

- The NetRS monitor which collects traffic statistics;
- And the NetRS controller which receives such statistics and periodically generates a placement plan for the RSNodes which then gets deployed.

The traffic metrics are extracted/computed from message packets metadata encoded in custom headers which is an agreed format within all intervening components (servers, clients, switches, etc.). The authors decided to make the format flexible to diverse replica

selection algorithms. One interesting header is the *Server Status* which characterizes the replica's computational load. these headers are also used to distinguish between key-value store traffic and others, which is not a problem in our case. With the gathered information, a selector will look into an incoming request and to the pool of replicas and perform a best match.

Although the introduction of newer in-networking hardware and the offloading of replica selection might make sense in a cloud-based environment where nodes are stationary, connected to reliable communication channels, usually organized in racks and following a hierarchical topology, it is not suitable for A MEC environment. We have to keep in mind that communication channels might not be wired and thus the introduced network hops would be more harmful than good. Moreover, we do not expect data center grade hardware at the edge, which implies that the cost of these in-networking devices would probably not be supported.

Therefore, we aim at the conventional scheme where each client can independently select a replica and make a request in a single hop.

2.1.5 Data Grids

Just like in MEC, dynamic grid architecture can have nodes join and leave the grid at anytime. This is a factor that is not considered in the previously analysed algorithms.

There are several replica selection algorithms directed towards dynamic grids, each basing its decisions on different sets of parameters. We start with replica selection techniques based on round trip time or distance. Under this category, we have Rigel [4] which selects the replica with smallest RTT. The system makes use of an NC Calculation System so that the nodes can periodically calculate their virtual coordinates and then these are stored with the node identified in a DHT so every node in the system has access to it. These virtual coordinates allow nodes to estimate their RTT to every replica without actually ever measuring them (i.e. issuing a request and waiting for a response). Although interesting, it is still a heavier solution than the ones previously analysed. Also, as we've seen before, "periodically"is the enemy of fresh and accurate.

For replica selection techniques based on response time, we have GRESS [22]. The response time is calculated based on the network parameters such as bandwidth and access latency. The best replica is predicted using historical log file that contains file transfer time, network status, server load and disk I/O information. Best replica is the one with minimum response time. The proposed lightweight replica selection module is based on Instance Based Learning (IBL) technology. IBL replica selection module consists of four processes, namely parameter setting, initialization of case-base, IBL replica selection, and update of case-base. During parameter setting, weights are assigned and contents of historical log file stored during initialization. IBL replica selection predicts which server is the best whenever a file is requested. The last process is used to update the case-base periodically. Again, this solution is too heavy for a MEC network and the necessary

information would take too long to spread.

There are also replica selection techniques based on some properties of the service such as Availability and Security, and others based on job time. These make heavy use of machine learning algorithms and thus are not even considered due to our battery constrained clients.

2.1.6 Final Considerations

Most replica selection algorithms directed towards distributed key-value stores posed as good candidates for MEC architectures for being lightweight and easy to operate on top of existing systems. There are some more naive and others more advanced. We decided to take a deeper look into C3 because it looks like the most complete and reliable of the group.

C3 is coined by some authors [3, 16] as the state-of-the-art algorithm for distributed key-value store replica selection. It is way more effective at reducing tail latencies compared to its competition, very efficient at preventing herd behaviors and very robust to performance fluctuations in the system. The C3 mechanism, however, is not trivial to understand and in spite of its major gains in tail latency, even the authors say that there's still a lot of room for improvement. Ultimately, it can be perfectly adapted to a MEC system and thus makes it a strong candidate for the baseline of our work.

L2 was also seen as an interesting alternative to C3. It is able to achieve similar results to C3 and is a lot simpler.

NetRS raised the bar once again, proving that there really is a lot of room for improvement on top of C3. Although attractive, we've seen that the algorithm is not suited for our target environments and that despite the improvements, C3 still poses as the most appropriate solution.

We've also looked at replica selection algorithms in the context of grid computing, where we've concluded that those algorithms are too computationally demanding for our needs.

In conclusion, C3 is what comes closer to what we are aiming to build. Even still, neither C3 nor any other algorithm we've seen account for mobile nodes where parameters such as battery percentage should weight on replica selection, as well as the possibility of intermittent availability or even sudden unavailability caused by churn. Also, because most of these algorithms assume reliable (wired) communication channels, they do not account for other bottlenecks that might arise in MEC environments, such as the AP being overloaded.

2.2 Replicated Storage at the Edge Systems

With the rise of the MEC paradigm and with data availability in mind, replicated storage systems started to emerge. Such replication might happen through various stationary

Table 2.1: Comparing Dynamic Replica Selection in Cloud Environments

	C3	L2	NetRS
Performance	Good	Good	Very Good
Complexity	High	Low	High
Uses Client-Observed Metrics	Yes	Yes	Yes
Uses Server-Sent Metrics	Yes	No	Yes
Uses Special Networking Hardware	No	No	Yes
Clients Can Select Replicas Independently	Yes	Yes	No

edge servers, the mobile clients themselves or even both. An example of such system is Thyme GardenBed [14], which actively replicates data objects through mobile devices in the same cell, passively replicates the same objects through mobile devices that download them from others cells, and such objects can even be replicated in the infrastructure servers if they become popular. Subscriptions are also replicated within the cells and the infrastructure. Regarding replica selection for data retrieval, however, Thyme GardenBed will always default to the infrastructure if the item is popular. Otherwise, the native Thyme methodology is used: iterating the list of replicas by their order of arrival, without any special criteria. Bellow we discuss some of the MEC systems we analyzed and a direct comparison can be found on Table 2.2.

EPHESUS [12], an ephemeral distributed data storage system for networks of handheld mobile devices, is inspired by use cases similar to Thyme's. This system doesn't employ edge servers and its storage substrate is restricted to mobile devices. It makes internal use of a Distributed Hash Table (DHT) to replicate the data so that content doesn't become unavailable as soon as their publisher leaves the network. To retrieve the data, the system first checks if it is already replicating such data, thus not needing to retrieve it from a peer; in case it doesn't, a system-wide (i.e. flood) search is performed. This is obviously not an optimal solution for our case because we are already considering one-hop messages.

MobiTribe [18, 19] makes use of an edge server as a proxy to redirect data requests. Since it is single-handedly managing the network traffic, it can redirect requests taking parameters such as load into account. However, this indirection already introduces extra latency to the network and the way it uses the edge server poses some serious scalability issues.

From the studied systems, none employed a sophisticated replica selection mechanism that takes into account network, load or device-specific metrics, either transmitted by the server or observed by the client, and that allows it to make an isolated pondered decision of which replica should be contacted.

Table 2.2: Mobile Edge Systems w/ Replication

	Thyme GB	Ephesus	GHT [9]	TOTA [5]	MobiTribe
Par./Full Replication	Partial	Partial	Partial	Full	Partial
Act./Pass. Replication	Both	Both	Active	Active	Both
Requires Infra.	Yes	No	No	No	Yes
Range	1-Hop	1-Hop	Multi-Hop	Multi-Hop	Multi-Hop
Network Structure	DHT	DHT	DHT	Unstruct.	Unstruct.
Rep. Select. Strategy	Static	Static	Static	Static	Dynamic

2.3 Mobile-to-mobile Communication

As mentioned previously, in case of congestion due to computational load, the load might not be on the replicas/servers but rather on the AP. In such cases, it might be attractive to open a direct communication channel between mobile devices, avoiding the infrastructure altogether. To this end, we've looked at possible communication protocols that cover this use-case and two stood out: TDLS and WiFi-Direct.

Although both of these protocols allow for mobile devices to establish direct links between them, they are not the same thing. TDLS requires that the devices are connected to the AP so that they can negotiate the direct connection. The protocol itself has mechanisms to evaluate if a better connection would result from opening this direct tunnel. TDLS operates in the background of the network and might automatically make the connection switch if it detects a performance increase. WiFi-Direct, on the other hand, enables a direct channel to be established while no WiFi network is available. It is similar to establishing a Bluetooth connection. WiFi-Direct is newer and struck us as being more programmer friendly.

2.4 Final Remarks

In this chapter we have looked into other computation environments, such as the cloud, to understand how their replicated systems adapt to the environment conditions and strive to select the replica that most positively impacts the overall system liveness for their data retrievals. From this analysis we have highlighted a particular *Feedback* algorithm, C3, which seems to be suited as the baseline for our developments.

We have then discussed some existing MEC systems which already employ replication to enhance their quality of service. When analysing their replica selection mechanism, however, we have seen that most employ a very naive strategy which never adapts to the system's and environment's variable conditions. The only exception to this rule was MobiTribe [18, 19] which uses a dynamic replica selection strategy; however, this strategy is coupled to a centralized piece of the system, the infrastructure, which is by itself a *hotspot* and single point of failure. Moreover, many of the discussed systems did not have a supporting infrastructure, which would make it impossible to adopt such strategy.

Finally, we have looked into alternative communication channels as a complementary solution to replica selection when constrains are detected in the communication medium, which is something to be expected more often than in cloud environments for instance, where communication channels are more reliable. From our brief discussion, WiFi-Direct poses as an attractive protocol.

In the next chapter we discuss Thyme GardenBed [14], the system we aim to enhance with our solution for experimental evaluation purposes.

Chapter 3

Thyme

3.1 Thyme

Thyme [11] is a time-aware reactive data storage system for wireless edge networks, that exploits synergies between the storage substrate and the publish/subscribe paradigm [13]. It combines the storage and P/S interfaces making their operations intertwined: for instance, the *insert* and *publish* operations are combined into one. Queries are in the form of subscriptions that have a specific time scope defining when they are active. Their active time-span can target the future, the present or even the past, expanding on the usual concept of *subscription* in a P/S system. Also, the reactive interaction model supporting the storage substrate avoids peers having to proactively search for content. Instead, it allows applications to react to new data being generated and stored. 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 relevant data is generated. 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 [13]. Because of time decoupling, custom operations were also added to this framework, such as the ability for a node to unpublish a previously shared item.

An asynchronous model comprised of mobile devices is considered. Nodes communicate by exchanging messages through a wireless medium and should be able to establish communication with their one-hop neighbors. Each node has a globally unique identifier. Nodes' clocks are also assumed to be synchronized (with a negligible skew).

Nodes are considered to be functionally symmetric, sharing the same responsibilities and having no particular roles, meaning that there are no centralized or specialized components, and each node can be a publisher, a subscriber, or both. Moreover, the area covered by the system is divided into square-shaped cells and all nodes inside a given cell collaborate with each other to form a virtual node. Cells can store (replicate) data items, metadata and subscriptions, thus acting as virtual P/S brokers. Over this network topology is the implementation of a DHT.

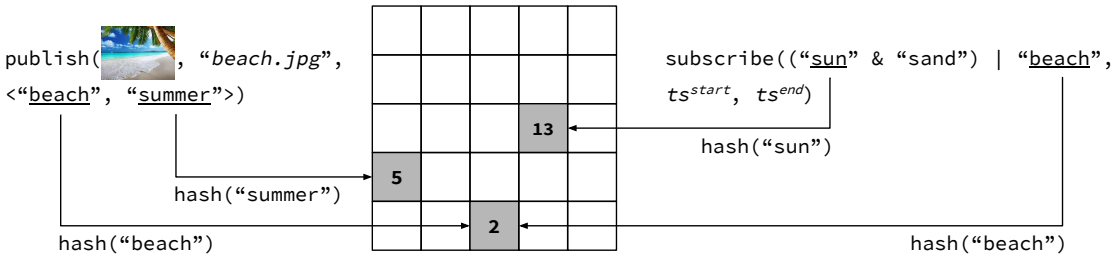


Figure 3.1: Example of Thyme's publish and subscribe operations. The tags' hashing determines the cells responsible for managing the object metadata (cells 2 and 5) and the subscription (cells 2 and 13). If a subscription has overlapping tags with a publication (and vice versa) it will also have overlapping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

Below we cover Thyme's main features:

- **Inserting / Publishing Data:** As stated before, the insert and publish operations are merged together. Each inserted data object has associated metadata. This metadata consists of:
  - the object identifier;
  - a set of tags related to the object (like hashtags in social networks);
  - a summary of the object (e.g. a thumbnail in case of a photo sharing application);
  - the insertion timestamp;
  - and the owner's node identifier.

  Thyme runs each tag though a hash function to determine which cell(s) should receive the object metadata. It then sends the metadata information to each of the determined cells, which will also be holding any subscription to those topics (tags). This way, the insertion of a data object into storage may trigger the sending of notifications to subscribers. Only the metadata is sent to save bandwidth and overall resources since the metadata tends to be much smaller than the actual data object. The data object is replicated by all the nodes of the publisher/owner's cell.

- **Replicating Data:** In order to provide data availability, Thyme has two replication mechanisms in place:
  - **Active replication:** Upon the insertion of a data object, said object is disseminated inside the owner's cell. From that point onward, every node inside the cell should be able to serve the data. This gives us the guarantee that the object will remain in the system even after the owner leaves. Furthermore, it allows for load balancing inside the virtual node. The object metadata is replicated in the same way inside the corresponding cell(s).

  - **Passive replication:** Nodes outside the owner's cell that retrieve the data object will also be replicating it (in a passive way) from the moment they receive it. This offers a better data availability and scatters the objects across the network.

  To fully take advantage of both mechanisms, Thyme adds a list of the object's replicas in the metadata.

- **Deleting Data:** The delete operation removes the object metadata indexed by the responsible cells, as well as the active replicas in the system. This causes the data object to be inaccessible to future subscriptions. This way, subscriptions targeting the past will not see deleted objects, even if these were initially available in the subscription's time frame.

- **Subscribing:** A subscription contains: its unique identifier, a query which is a logic formula composed of conjunctions and disjunctions of literals (that will match the tags), a timestamp indicating the beginning of the time frame for the subscription, another timestamp indicating the upper bound of the time frame for the subscription, the identifier of the node making the subscription and the identifier of the cell where the owner is located. Following the same method as in the insertion, the tags (this time inside the query) are hashed to obtain the corresponding responsible cells. The subscription will be registered within those cells, effectively triggering data notification(s) if there are already data objects with corresponding tags and the subscription spans to the past (far enough to cover the matching data objects' timestamp). From this point onward, when the cells responsible for the subscription receive object metadata that matches it, they will send such metadata to the subscription owner. The subscription owner will be responsible to retrieve the data object from one of the replicas listed in the metadata or ignore the subscription altogether. The unsubscribe operation is as simple as a message asking the responsible cell(s) to delete the stored subscription.

- **Retrieving Data:** The client will choose a node from all the replicas in the replication list and send a retrieve request for the desired object. If a negative reply is received, the requester proceeds and tries the next replica in the list (until no more options are available, or a maximum of retries is reached). If the maximum number of tries is reached trying passive replicas, it will try to force the download from the active replica before as a last resort.

A node with the intention of joining the system will wait a configurable amount of time for a beacon sent by a neighbor in the same cell. If such beacon is received, it can be used as an entry point. At this point the node will send a join request and, if successful, it will receive the cell state. If a maximum number of retries is reached, the node assumes it is alone in the cell, and starts operating normally.

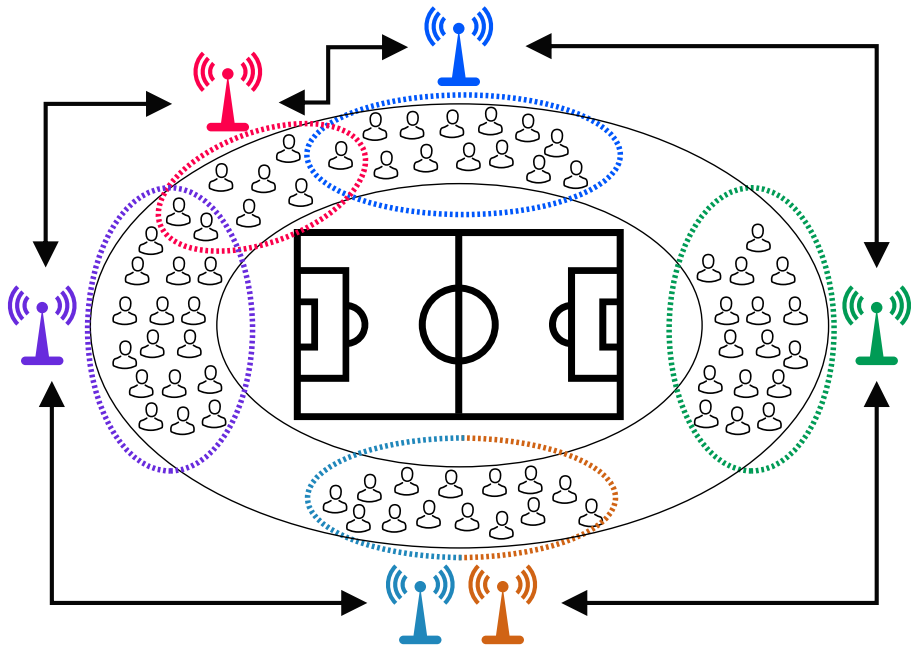


Figure 3.2: Application in the context of a football stadium. Adapted from [14].

3.2 Gardenbed

Gardenbed [14] is a framework that leverages stationary nodes within the edge infrastructure to provide a persistent publish/subscribe system to a set of mobile devices, distributed across multiple network regions. It leverages both device-to-device and device-to-edge interactions with the goal of optimizing access to popular data, and allow such data to be available to all users, hence creating a persistent and global end-to-end storage and dissemination network.

Infrastructure-wise, each edge server can be connected to multiple wireless access points, each of which is responsible for managing its own region. Each AP can connect only to one server. Moreover, each mobile node belongs to a single region, even though it may be in range of multiple APs. This is depicted in Fig. 2. These servers connect multiple regions through a caching and prefetching mechanism. Therefore, mobile nodes are able to retrieve data that was generated in others regions. However, nodes from different regions won't have access to all the data in the other region: there will be popularity metrics in place to decide which data gets exposed. These servers also allow for the mobile nodes to offload some of their management responsibilities in their region.

Gardenbed offers an interface to the mobile nodes to communicate with the servers, comprised of the following main operations: **publish**, **unpublish**, **subscribe**, **unsubscribe** and **download**. There are additional operations such as **isOnTheEdge**, which indicates if a given data object is being cached by the edge server. Clients map their operations into operations of this API and the data-related operations performed within the region will be periodically batch-disseminated to the server. On the other side, the

server will be notifying clients of newer data or other changes. Mobile nodes should listen to the following commands from the server:

- **notification** of a new publication from a remote region;
- **update** of a given object's metadata to keep it consistent with the server;
- **download** of a given object the client is holding, in order for the server to cache or serve it to a client in another region;
- **unpublish** of a given object so that it is effectively removed from the system.

Gardenbed supports the idea of clusters (which can be seen as a virtual node comprised of several mobile nodes). In these cases, the interaction from client to server can be delegated to the cluster-head node. To define a cluster-head, the Gardenbed server API also offers the **setClusterHead** operation.

The framework offers some hooks for the programmer to register their logic. One such case is the Popularity algorithm, which determines what data items must be uploaded to Gardenbed. Gardenbed asynchronously collects and caches the most popular items within the region, according to the injected popularity algorithm. Items downloaded and cached in this fashion are stored in the Local Popularity Cache (LPC). With this, we get to 1) serve subscriptions from other regions and 2) same region nodes can grab the data directly from the edge instead of requesting it from their peers, which drops the number of download requests to mobile clients effectively saving battery life. The item will now be registered as being on the edge and is up to the cluster-head node to disseminate the information to its peers. This process is run periodically and thus, the LPC will continuously be updated with what is most popular.

Another piece of logic that can be registered by the programmer is the Matching Logic algorithm, which will tell Gardenben how to match publications with subscriptions. With this, the server will match every subscribe operation retrieved from the *cluster data* against the data in its caches. For every match, it will trigger a *notification* to the subscriber. Because the subscriptions are potentially relayed and stored in both the region peers and the server, data might flow from both ends. To avoid duplicates and other shortcomings, the programmer should also register a Notification Priority Policy algorithm that determines whether a notification should be sent by the edge, by the underlying system or by both.

The subscribe operations extracted from the *cluster data* are also used to build the *subscription catalog*, which consists in a set of subscriptions from the region that will be sent to other regions so these know their interests. This catalog is periodically recomputed, avoiding the need for the server to keep state information about individual nodes. Fig. 3 depicts this cross-region P/S process.

This cross-region process makes use of two other caches:

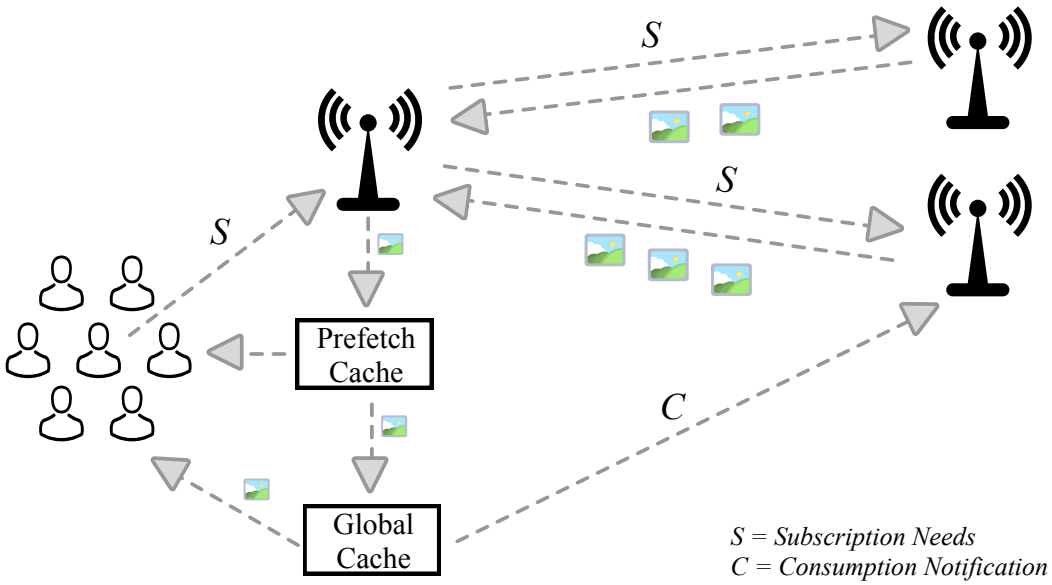


Figure 3.3: Global P/S execution process. Adapted from [14].

- **Prefetch Cache:** Stores data and metadata from other regions. The turnover rate of the contents in this cache is expected to be quite high, increasing with the total number of nodes in the system;
- **Global Cache:** Stores entries from the prefetch cache that were specifically downloaded (i.e. considered relevant), so that these are stored in a more persistent way and available to other users. This is an optimization done under the assumption that geographically-adjacent users share similar interests to some degree.

The process happens in three steps (Fig. 3):

- **Dissemination of Subscription Catalog:** As stated before, each server periodically broadcasts its clients' *subscription catalog* to other regions' servers.
- **Provisioning the Subscription Catalog:** The server looks at the received *subscription catalog* and its own LPC, and tries to match cache entries to the remote subscriptions. If there are any matches, the data and metadata will be sent in batches. Although this means that only popular data is transmitted to other regions, it translates to great bandwidth savings. Moreover, it makes use of additional information that comes attached to the **subscription catalog** in order to make this step much more efficient in terms of network utilization. When matching LPC entries to the *subscription catalog*, the Matching Logic algorithm is once again applied so we are sure that only the relevant matches are considered.
- **Notification of Remote Publications:** Upon the reception of a periodic *data provisioning message*, a server notifies the corresponding subscribers of the arrival of the data, and stores the received items in its Prefetch Cache. These will remain there until they are moved to the Global Cache or evicted by the arrival of new data.

To retrieve content, a client sends a download message to the server. The operation will explicitly state if it's a local or remote download (the information can be retrieved from the metadata). For a local download, either the server serves the item from one of its caches or it forwards the request to the mobile replicas within the region. For a remote download, the server will first look for the item in its Prefetch and Global caches. If the item is found it is send to the client, otherwise the server will reroute the request to the source region. There, that region's server will look for the item within its caches and, if it isn't there it will forward the request to the mobile replicas within the region. When a server receives the reply to a remote download with data from another region, before routing the requested item to the end-user, it will proactively cache the incoming data in its Global Cache.

Unpublish operations have a global scope, since the item to unpublish may have crossed the boundaries of the current region, and may be cached in several servers or even be indexed in multiple regions. Thus, whenever a server receives an unpublish operation it deletes the entries associated to the referenced item in all of its caches (if present). Moreover, except the server of the region emitting the operation, they also propagates the operation to the cluster-head responsible for handling the item's metadata, for it to execute the operation locally [14].

3.3 Thyme Gardenbed

GardenBed is able to enhance Thyme in a way that allows for inter-region communication (something that is not possible using Thyme as a standalone system). Fig. 2 greatly illustrates the concept.

A Thyme cell can be seen as a cluster for GardenBed. By leveraging on this organization, a cluster-head is picked per cell, making it responsible for interacting with GardenBed. All traffic targeting a cell is sent to the cell's cluster-head, and then disseminated inside the cell. This approach trades-off the over-utilization of a single node's resources within each cell for the resources needed to proactively keep cell membership in the server. To elect the cluster-head, a *stability index* is calculated. This index is derived from local hardware information such as battery percentage and represents the probability of a node leaving the network either by leaving the covered geographical area or by shutting down. Each node is capable of comparing its index to its neighbours and if is the one with higher stability it will take the initiative. Note that a new cluster-head might be elected at each data dissemination cycle.

The metadata for the objects is also extended to contain the boolean flag *onTheEdge*, which is set when the object is uploaded to the server and updated when proper.

Following the above section, when an object that originated in region A is downloaded for the first time in region B, the item will be moved from the server's Prefetch Cache to the Global Cache, which will guarantee its availability for all other mobile nodes in region B. The object is also indexed in Thyme so that it can be retrieved from other devices.

Regarding the **matching logic**, it is two-fold:

- Matching subscriptions against publications: all cache entries related with the subscription tags are retrieved and then filtered to keep only those within the subscription time frame;
- Matching publications against subscriptions: Filters the subscriptions list to retrieve only those matching the publication.

The configured **notification priority policy** is to always prioritize the usage of the edge servers in favor of the mobile clients, meaning that whenever a subscription is matched for an object which metadata is flagged **onTheEdge**, the cell trusts that the edge server will send the notification and will thus refrain from sending one itself to avoid any duplicates. This strategy is aimed at reducing battery consumption because now mobile devices will only trigger notifications of data not available in any of the server's caches (i.e. not popular).

Finally, the **replica selection** policy follows the same logic as the **notification priority policy**, which is to favor the edge servers in order to preserve the battery capacity of mobile nodes. If the object is **onTheEdge**, then the mobile devices will **always** favor the edge server over the mobile replicas. This seems like a decent strategy and is most likely the easiest to implement, but it can ultimately incur in higher latencies and system congestion depending on network load. If we consider that the majority of traffic generated in the system is related to the most popular content, then it's easy to understand that the higher the mobile nodes count in the system, the faster the edge server's and network devices' resources will get saturated. If nodes could share information between them and the edge which would allow them to detect or even prevent these scenarios, then such information could be used to make better informed replica selections and allow the system to scale more gracefully. With this thesis we understand and decide on these metrics, and take them into account to enable mobile nodes to independently decide which might be the best replica to download the object from. This is done in order to minimize communication latencies and effectively distributing load whilst being conscious about mobile devices power consumption.

3.4 Final Remarks

In this chapter we have discussed Thyme, a time-aware reactive data storage system for wireless edge networks. We have described Thyme's replica selection policy, which iterates through the available replicas in no logical order; when the selected replica fails to provide the desired file, it tries the next one until it has tried a maximum number of passive replicas, and then makes a last attempt by contacting the cell or if it has previously tried the cell and it failed to provide the file, which the node assumes is permanently gone from the system. The idea is that we should try to move the load away from the active

replication cluster as it most likely serves the most downloads regarding that file. Also, files that cannot be found in their active replication cluster are assumed to be *unpublished*.

We have also discussed GardenBed, a framework that leverages stationary nodes within the edge infrastructure to provide a persistent publish/subscribe system to a set of mobile devices, and Thyme GardenBed, where Thyme is enhanced by GardenBed's infrastructure server. Here, the replica selection policies shifts to give preference to the infrastructure. If the infrastructure is not available, or fails to provide the required file, we fallback into Thyme's replica selection policy. The idea is that the infrastructure server is a more powerful node which is not power constrained, thus it should serve most of the downloads. This is a valid strategy which yields interesting results compared to its implementation complexity; however, it is still not flexible (dynamic) and all nodes will follow these rules independently of system performance and network congestion, which might severely degrade the quality of service in some scenario where it would have been preferable to trade some of the mobile devices' battery capacity for liveness.

In the following chapter we present Wasabi, our replica selection algorithm that is tailored for MEC systems. With this, we discuss each of the concerns raised by the practice of replica selection and in special, those that are characteristic of MEC environments and how we tackle each of them. We also present the replica selection behind it and how the programmer can integrate it within their system. Moreover, we discuss how we have integrated Wasabi into Thyme GardenBed.

Chapter 4

Proposed Solution

As we have introduced in Section 1.4, to accurately answer the question *"Where should I retrieve the object from?"*, given a set of available replicas, we need to have **fresh** information about those replicas' state and potentially even be able to *predict* network conditions. To have such information available to us, we need a pluggable system that can put and retrieve these metrics to and from the transport layer, ideally in a non-evasive fashion.

In this chapter we first give a detailed overview of the designed replica selection framework. Then, we explain the replica selection strategy/algorithm we found to be optimal for MEC systems. And lastly, we describe our framework integration with Thyme GardenBed [14].

4.1 Overview

Starting our design we understood that, in essence, the problem boils down to a **client** having interest over a piece of data that is owned by one or more **server**s, independently. We have seen from Section 2.1 that there are three categories of replica selection algorithms and that these build on top of each other. *Feedback* is the most comprehensive because it makes use of both metrics sent by the servers and metrics perceived by the client. We have thus decided to design a system that enables *Feedback* strategies for replica selection, as it in turn enables us to build any of the others as well simply by omitting unnecessary components (e.g. not sending any metrics from servers and only considering metrics measured by the client, which classifies the underlying ranking algorithm as *Client-Independent*).

Feedback strategies also bring another performance idea to the table, which is that the extra information sent by the servers to the clients through the transport layer does not necessarily need to generate new messages within the system. Instead, it can use already existing messages and *piggyback* metrics onto those. This is what we have done with Thyme GardenBed [14], as we will explain later in this chapter.

Our first step was thus to split our architecture in two independent parts: client-side and server-side. Each part provides a disjoint set of components and these can coexist at

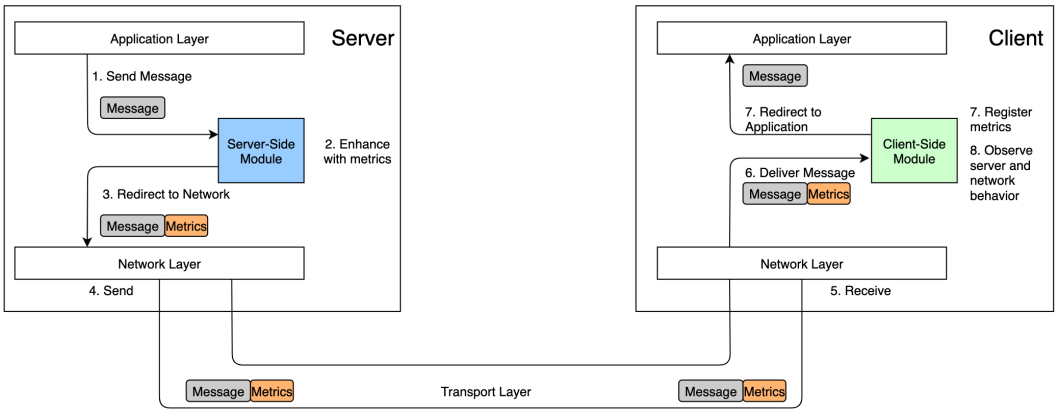


Figure 4.1: Replica Selection Feedback System as an Application/Network middleware. The server sends extra information regarding its internal status with the system messages and the client stores those metrics plus some extra observed information. The client will then use this information to pick the most appropriate replica for each of its operations with a certain degree of certainty.

the same runtime without any conflict - this is to cover the scenarios where the mobile nodes are functionally symmetric (both client and server), as is the case with Thyme [13].

Then we had to think about component placement. From the beginning of this document we have proposed a middleware service that could seat between the Network and Application layers to transparently include and consume the included metrics packets on the server- and client-side, respectively. Figure 4.1 shows how the metric dissemination flow of a system integrating our framework should unfold. In a nutshell, the server-side component will proxy the Network layer (1), trapping all messages ready to be sent in order to include extra information (its metrics) (2). It will then forward the message to the network (3). On the client-side, there's another module now proxying the Application layer which will first pre-process the delivered messages to extract the previously injected metrics (7). It should also take the opportunity to take some measurements on the network behavior, update some existing state related to that server or generate any other metric value that is required by the replica selection algorithm (8). Finally, the original message is delivered to the Application layer.

We also need to look at it from the opposite direction, that is, a client request to a server. Whenever we need to contact a server and are presented with multiple options, we should make use of the previously gathered metrics to decide which server to contact. Figure 4.2 depicts this flow: the application dispatches the request which is proxied by the client-side module (1) to add the destination address, which will be the replica we perceived as being the most favourable (2). The request is now redirected to the Network Layer (3) as it was intended to be sent to the selected server.

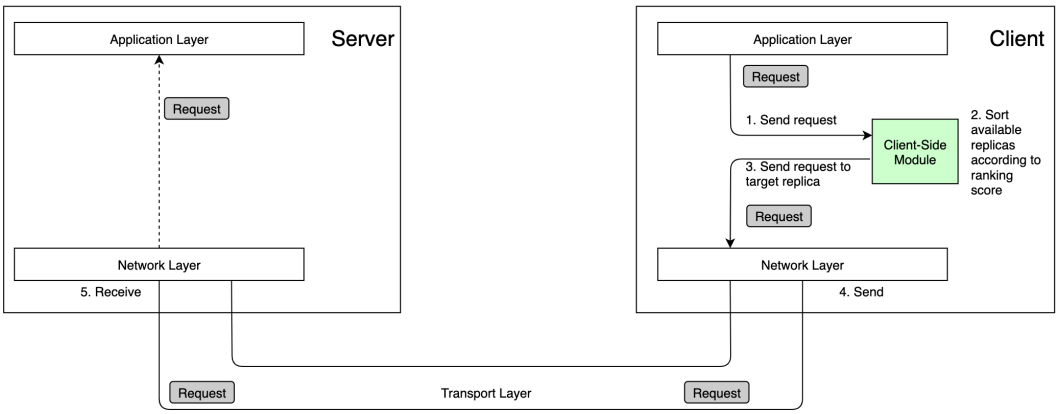


Figure 4.2: Replica Selection Feedback System as an Application/Network middleware. The client relies on the Replica Selection module to pick a request target before pushing such request into the Network.

4.2 Proposed Framework Architecture

Having an high-level overview of how the framework should operate, we will now dive in detail into each of the modules' components, the available interfaces as well as some out-of-the-box implementations and how the programmer can use them.

4.2.1 Server-Side Components

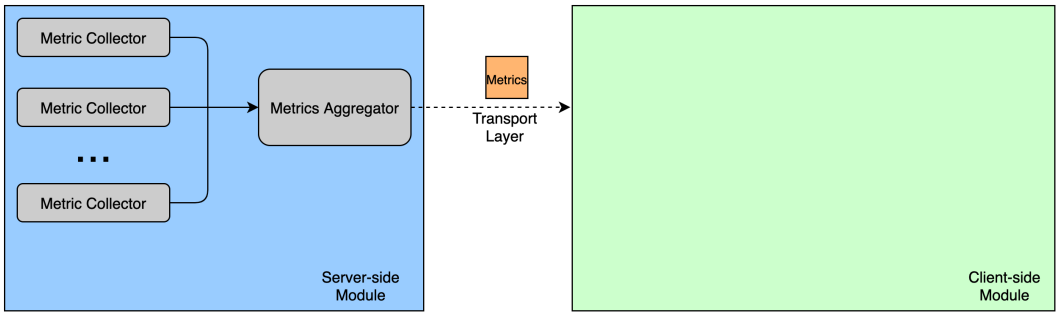


Figure 4.3: Inside the server-side module

4.2.1.1 Metric Collector

A *Metric Collector* is a small footprint component that contains the necessary logic to read or compute a system metric. This logic is to be provided by the developer. As a basic - and most common - use case, a Metric Collector will be a purely functional component, i.e., it will perform stateless computations to produce a value. For such scenarios we can directly create instances of the interface in Listing 4.1, which are inherently thread-safe. However, for more complex scenarios where we might want to provide a stateful implementation, care should be taken to avoid concurrency problems. One use case that we have found to be common is to have a Collector that stores value samples and returns a

value based on the stored samples when queried for it. For these we also offer the class in Listing 4.2. This class already offers a synchronized implementation of the *collect* method (Line 9), as well as a synchronized method to add sample values (Line 18). And finally, as the example of a concrete implementation, we have the class in Listing 4.3 which should be used when we want to compute the average of the stored values and use it as the metric value.

```
public interface MetricCollector {
    Double collect();
}
```

Listing 4.1: MetricCollector

```
1  public abstract class SampleMetricCollector<T extends Number> extends
   ↪  SynchronizedMetricCollector {
2      private final Number[] samples; // Sample vector
3      private final int maxSize;
4      private int currentIndex = 0;
5      private int sampleCount = 0;
6      private final T defaultValue;
7
8      @Override
9      protected synchronized Double collect() {
10         Number[] currentSamples = sampleCount >= maxSize ? samples :
           ↪  Arrays.copyOfRange(samples, 0, currentIndex);
11         DoubleStream samplesStream =
           ↪  Stream.of(currentSamples).mapToDouble(Number::doubleValue);
12         return aggregateSamples(samplesStream)
13                 .orElse(defaultValue.doubleValue());
14     }
15
16     protected abstract OptionalDouble aggregateSamples(DoubleStream samples);
17
18     public synchronized void addSample(T sample) {
19         samples[currentIndex] = sample;
20         currentIndex = (currentIndex + 1) % maxSize;
21         sampleCount++;
22     }
23 }
```

Listing 4.2: SampleMetricCollector

```
public class AverageMetricCollector<T extends Number> extends
↪  SampleMetricCollector<T> {
    public AverageMetricCollector(int maxSize, Number defaultValue) {
        super(maxSize, defaultValue);
    }

    @Override
    protected OptionalDouble aggregateSamples(DoubleStream samples) {
        return samples.average();
    }
}
```

Listing 4.3: AverageMetricCollector

4.2.1.2 Metrics Aggregator

The *Metrics Aggregator* is the component that aggregates all the metric values produced by *Metric Collectors*. To be able to collect their values, Metric Collectors have to be registered into the Metrics Aggregator in association with the respective Metric label. When queried, the Metrics Aggregator will poll the Metric Collectors for their values and will store them in a key/value data structure to which it will then return a reference. This data structure is already the format expected to be processed by the client-side. Listing 4.4 represents the public interface for the Metrics Aggregator component.

The programmer should first register the Collectors at bootstrap time. The Metrics bundle should be retrieved any time we want to send metrics to the client, as shown in Figure 4.1. Although we offer simple implementations for the *MetricsAggregator* interface, we do not offer any component that automatically performs the concatenation of the message with the metrics and serialization of such.

```
public interface MetricsAggregator<Metric extends Enum<Metric>> {
    Metrics getMetrics();
    void addCollector(Metric metric, MetricCollector collector);
}
```

Listing 4.4: MetricsAggregator

To sum up, the programmer should:

1. Create a MetricsAggregator component specifying the metric enumeration;
2. Create a MetricCollector for each server-side metric and register it with the respective Metric within the MetricsAggregator;
3. Join the application serializable message with Metrics in order to serialize everything according to the marshalling protocol in place and send it to the network.

4.2.2 Client-Side Components

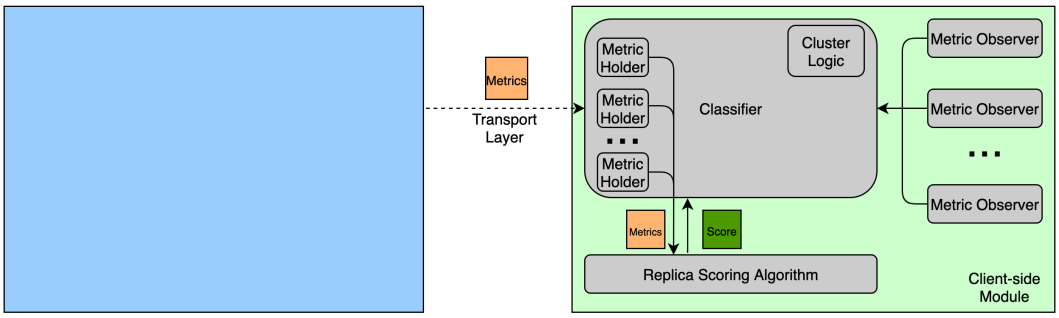


Figure 4.4: Inside the client-side module

4.2.2.1 Replica Classifier

The Replica Classifier is the central piece on the client-side. As the name indicates, it is responsible for classifying the available replicas for any given operation. It does so by taking a set of replicas and sorting them from best to worst using the collected metrics. The certainty on that ordering is proportional to how many metrics we have over that set of replicas and how fresh that information is. But the Classifier does more than this. It stores the metric values for each (already) known node with different retention, batching and even decaying policies.

```
1  public interface ReplicaClassifier<Replica extends Comparable<Replica>, Metric
   ↪  extends Enum<Metric>> {
2
3      Set<Replica> sort(Collection<Replica> replicas);
4
5      void record(Replica replica, Metrics metrics);
6
7      void record(Replica replica, Metric metric, double value);
8
9      void addMetric(Metric metric, Class<? extends MetricHolder> metricHolder);
10
11     void addMetric(Metric metric, Class<? extends MetricHolder> metricHolder,
       ↪  MetricDecayFunction decayFunction);
12
13 }
```

Listing 4.5: ReplicaClassifier

Listing 4.5 contains the public interface for the Replica Classifier. Other than sorting a collection of replicas, it allows for the record of individual or bundled metrics, to configure how the values are stored internally (e.g. only keep last, keep a moving average,

etc) and to define a decaying policy for the metric. It is important to note the generic types declared on the interface. The *Replica* generic placeholder represents the nodes' identifier, i.e., what identifies an individual node/server. Here we can use any kind of comparable data structure and it will be used to index information on that node, retrieve it and guarantee that we do not consider the same node more than once. This can also be exploited for more complex use cases, for instance, we could identify a node within the module not only by its id but rather by the combination of its id and the communication protocol. This would allow for the same node to be considered more than once but each time with a different protocol. This can be useful to explore the alternative channels proposed in Section 2.3.

The *Metric* generic will usually match the metrics' set representation of the server-side module. However, because in a distributed system sometimes there's different node versions and even different types of servers, the client-side module will simply ignore any metric that it does not recognize. Moreover, we can have different *Replica Classifier* instances in the client in case we have interest in different kinds of servers which emit different sets of metrics.

We already offer an out-of-the-box standard implementation of this interface, *StandardReplicaClassifier*, which should fit most use cases and already uses data structures from Java's concurrent package to avoid unwanted interference. However, if there is any required customization for this component, then the programmer will still be able to easily extend either the provided implementation, since its internal state and important inner methods are all accessible to extending classes, or the provided interface if they required a fundamental change in the functionality.

The *Replica Classifier* sorts the replicas based on their score, using a custom *Comparator* that sorts elements from highest to lowest score. The score is calculated by the underlying *Scoring Algorithm* which contains the formula that is applied over the metric values. We will cover this component, as well as the *Cluster Logic* in further sections. These can be ignored for now. However, it makes sense to discuss the *Decay Function* in this context.

Line 11 of Listing 4.5 shows that we can also register a *MetricDecayFunction* for a given *Metric*. This decay function will take into consideration the elapsed time since we have last record a value for the given *Metric* and will penalize the metric according to said time. Because the *Scoring Algorithm* always has a scale (i.e. it only produces scores between a configurable minimum and highest value), there is also the concept of the neutral value, which usually will translate to the middle value on the scale. This value is used as the score for the replicas we do not know anything about (i.e. to which we still hold no metrics). This is a way to neither penalize nor favor unknown outcomes. Having this into consideration, we felt like we needed to addresses the cases where we would score a replica above the neutral score if no decay were applied to its metrics, but, with decay, it goes bellow said threshold. In that situation, we consider the neutral score as the replica's score instead. Otherwise, we always consider the decay score. And, when no

decay function is applied, there is no change to any metrics' value over time. This is our integrated mechanism to deal with metrics freshness.

The *Replica Classifier* does not hold the metrics directly but rather stores them within containers (*Metric Holders*) that deals with further concerns regarding the stored value. To be able to consider a given metric, we must register it at bootstrap time using the *addMetric* method (Listing 4.5, line 9 and 11) where we provide the reference to an implementation of *MetricHolder*.

In conclusion, the programmer should create the Classifier at system bootstrap time specifying the node's identifier type (the *Replica* generic), the *Metric* enumeration, and configure it with a *Scoring Algorithm*, *Cluster Logic*, *Metric Holders* and *Decay Functions*, when applicable. Some of these components still were not discussed, so that is what we will keeping doing throughout this section.

4.2.2.2 Metric Observer

In spite of having Metric Collectors on the server-side, we still need a mechanism to produce metrics on the client-side. We thought about using Metric Collectors for this purpose as well, however, Collectors are pull-based and we required push-based component. Hence, we have introduced Metric Observers, which, like Collectors do hold programmer-provided logic to compute a metric value in the system. The difference is that these do not need to be registered within the framework. Instead, these require a reference to the Replica Classifier in order to push their readings. The push model is required to update the stored information regarding a given node on system events, such as network activity. One simple example extracted from C3 metrics is the measurement of the server response time, to which we first need to save the request id with a timestamp and, when we eventually receive a response, we measure the elapsed time and push this value to the Classifier (Listing 4.7).

```
1  @RequiredArgsConstructor
2  public class MetricObserver<Replica extends Comparable<Replica>, Metric
   ↪  extends Enum<Metric>> {
3
4      protected final Metric metric;
5      protected final ReplicaClassifier<Replica, Metric> replicaClassifier;
6
7      protected final void updateMetric(Replica replica, double value) {
8          replicaClassifier.record(replica, metric, value);
9      }
10 }
```

Listing 4.6: MetricObserver

For this component(s) we provide a base class (Listing 4.6) that should be extended by

the developer. As with the Replica Classifier, we need to provide the actual type definition for the nodes' identifier and the metrics enumeration. Additionally to the Response Time Observer, we also offer the *Outstanding Requests Observer* out-of-the-box to track how many requests we have outgoing to a server and to which we have not received a response yet.

```
1  public class ResponseTimeObserver<Replica extends Comparable<Replica>> extends
   ↪  MetricObserver<Replica, C3Metric> {
2
3      private final Map<Replica, Map<Object, Long>>
       ↪  replicaRequestIssuingTimestamps = new ConcurrentHashMap<>();
4
5      public void registerRequest(Replica replica, Object request) {
6          Map<Object, Long> timestamps =
           ↪  replicaRequestIssuingTimestamps.computeIfAbsent(replica, r -> new
           ↪  ConcurrentHashMap<>());
7          timestamps.put(request, System.currentTimeMillis());
8      }
9
10     public void computeResponseTime(Replica replica, Object request) {
11         Map<Object, Long> timestamps =
           ↪  replicaRequestIssuingTimestamps.computeIfAbsent(replica, r -> new
           ↪  ConcurrentHashMap<>());
12         Long timestamp = timestamps.remove(request);
13         Optional.ofNullable(timestamp).ifPresent(startTime -> {
14             long latency = System.currentTimeMillis() - startTime;
15             updateMetric(replica, latency);
16         });
17     }
18 }
```

Listing 4.7: ResponseTimeObserver

4.2.2.3 Metric Holder

A Metric Holder (Listing 4.8) is simply a container for the metric current value. These are needed because different metrics require different ways to be stored; we might just want to keep a discrete value or we might need to keep something more complex such as a moving average. We already offer a variety of Metric Holders out-of-the-box. These are used to configure the value storing policy for each metric on the Replica Classifier (method *addMetric* in Listing 4.5).

```
public interface MetricHolder {

    Double getValue();

    void put(double value);

    Long lastRecordedTimestamp();

}
```

Listing 4.8: MetricHolder

4.2.2.4 Replica Scoring Algorithm

The Replica Scoring Algorithm is the component that contains the ranking formula. It takes the set of metrics and computes a score. As a guideline, the better the metrics, the higher the score should be. The Scoring Algorithm component consists of the interface on Listing 4.9. As mentioned in one of the previous sections, the algorithm should produce scores within a scale (i.e. only produces scores between a configurable minimum and highest values). For this, the programmer should follow the formula below when implementing the *score* method:

$$score = min(MAX\_SCORE, max(MIN\_SCORE, formula(metrics)))$$

We also offer an abstract class with this implemented out-of-the-box, *AbstractScoringAlgorithm*, which might be easier to extend. Still, this is just a guideline and not required for the system to function properly. In some cases it might not even make sense to set score bounds and thus we still offer the interface. Aside from this, we also offer some implemented scoring formulas, such as C3 (Listing 4.10) and our own formula which deals with more MEC specific concerns and which we will cover in a section ahead.

```
public interface ReplicaScoringAlgorithm {

    double score(Metrics metrics);

    double getMinScoreValue();

    double getNeutralScoreValue();

    double getMaxScoreValue();

}
```

Listing 4.9: ReplicaScoringAlgorithm

```java
public class C3Algorithm extends AbstractScoringAlgorithm {

    // omitted source code...

    @Override
    public double algorithm(Metrics metrics) {
        Double queueSizeBoxed =
        ↪  metrics.stringifyKeyAndGet(C3Metric.QUEUE_SIZE);
        Double serviceTimeBoxed =
        ↪  metrics.stringifyKeyAndGet(C3Metric.SERVICE_TIME);

        if (queueSizeBoxed == null || serviceTimeBoxed == null) {
            return neutralScoreValue;
        }

        double queueSize = queueSizeBoxed;
        double serviceTime = serviceTimeBoxed;

        double outstandingRequests = Optional.ofNullable(
        ↪  metrics.stringifyKeyAndGet(C3Metric.OUTSTANDING_REQUESTS))
            .orElse(0.0);
        double responseTime = Optional
            .ofNullable(
                metrics.stringifyKeyAndGet(C3Metric.RESPONSE_TIME))
            .orElse(serviceTime);

        double concurrencyCompensation = outstandingRequests *
        ↪  concurrencyWeight;
        double queueSizeEstimate = pow((1 + concurrencyCompensation +
        ↪  queueSize), QUEUE_FACTOR);
        double latency = (responseTime - serviceTime) + (queueSizeEstimate *
        ↪  serviceTime);

        return maxScoreValue - latency;
    }
}
```

Listing 4.10: C3Algorithm

To use the algorithm the programmer simply has to wire it through constructor injection if they are using the *Standard Replica Classifier*, or make sure they sure the *score* method of the *Scoring Algorithm* to compute a score for the replica and use that value to sort the available replicas. As a final note regarding implementation details, it should be noted that even though we operate with Java Enums regarding metrics, which does not allow us to extend a set of existing metrics, we can still circumvent this because the

Replica Classifier should use the String version of the enumeration items to communicate the metrics to the Scoring Algorithm, which means that we can use already implemented algorithms that use a given Enum underneath by simply declaring the same items with the exact same name on our new Enum (e.g. if we create a new metrics Enum with the same items as *C3Metric* in Listing 4.10: *QUEUE_SIZE*, *SERVICE_TIME*, etc, then we can use the existing C3 algorithm implementation).

4.2.2.5 Cluster Logic

Finally, because some systems might have the need to tell apart cluster nodes from single nodes when scoring them, we have introduced the Cluster Logic component. These systems can be, for instance, cluster-based DHTs on which clusters of nodes act as virtual nodes with their own role in the system.

This is a component that hooks into the Replica Classifier and requires two developer-provided functions:

1. A predicate that indicates whether or not the current node is a cluster;
2. A function that retrieves the cluster nodes from the known replica set on the Classifier.

This is an optional component. However, if provided when using the *Standard Replica Classifier*, it will classify a cluster with the average score of all its composing (known) replicas.

To register the Cluster Logic the programmer should implement the interface in Listing 4.11 and then inject it into the Replica Classifier through constructor injection when using the *Standard Replica Classifier* or, when using a custom implementation, simply make use of it to compute the clusters' score.

```
public interface ClusterLogic<Replica> {

    boolean isCluster(Replica replica);

    Set<Replica> findClusterNodes(Replica cluster, Collection<Replica>
    ↪  replicas);

}
```

Listing 4.11: ClusterLogic

4.2.3 Summary

In this chapter we have presented our replica selection framework architecture. We have separated it in two modules: server- and client-side; and described each of the modules

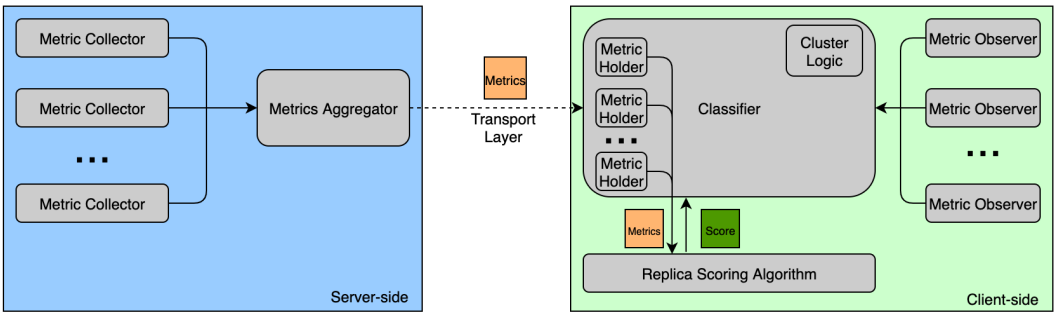


Figure 4.5: Replica Selection framework architecture diagram. The server produces a metrics bundle that is sent over the network to the client. The client forwards this bundle to the classifier to store information about this server. The metric holders will store the value of the corresponding metric. Metric Observers also produce individual metrics, on the client. When we need to sort a set of replicas we pass the stored information on each replica to the scoring algorithm which in turn will produce scores. The replicas are sorted by their descending scores.

components in detail and what role they play in the system. We have also discussed how these components should be placed. We explained how these components could be used and the available APIs. Furthermore, we have discussed some component specific challenges such as how to deal with concurrency and freshness, and presented some of our out-of-the-box implementations. Additionally, we have discussed some possible scenarios and advanced usages show as how the framework could be harnessed to support multiple (alternative) communication channels. Figure 4.5 gives the full picture of the framework.

4.3 Replica Selection Strategy for MEC Systems

4.3.1 Picking a baseline

From our analysis on Section 2.1, two options stood out as possible baselines to our work: C3 (Section 2.1.2) and L2 (Section 2.1.3). We first looked at L2 and soon realized it is not a suitable baseline for our work because it only takes into consideration client-side measured metrics: the outstanding requests to a server and the perceived response time. Even though L2 can achieve a similar best performance comparatively to C3 in cloud environments, as concluded by the authors of [3], MEC environments are in contrast more volatile and present additional challenges. Here, communication channels are less reliable and the available replicas are constantly varying. Therefore, it is best to use a *Feedback* algorithm when considering a replica selection strategy for a MEC system, in order to get information both from the client-side as well as the server-side. This allows us to not only decide upon the explicit values but also infer new information, e.g., we can compare reported server-side values with perceived client-side values to be able to distinguish server resource saturation from a network congestion and thus be able to

consider alternative communication channels. C3 is thus a better baseline for our use case.

C3 is a state-of-the-art [16] *Feedback* algorithm that combines a replica ranking scheme with rate-control and backpressure. The replica ranking scheme is a mathematical formula that computes a server's expected latency given the server's reported queue size and average service time, the client's pending requests and the observed server response time. The formula is as follows:

$$\psi_s = R_s - \frac{1}{\bar{\mu}_s} + \frac{(\hat{q_s})^3}{\bar{\mu}_s}$$

where ψ_s is the expected latency for server s, $\hat{q_s} = 1 + os_s \cdot w + \bar{q}_s$ is the queue-size estimation, os_s is the number of pending requests the client has to s, w is a configurable concurrency weight number that represents the potential number of clients in the system (in a cloud system this might be an exact number, like the number of nodes in a cluster, but in a MEC system it has to be a sensibly configured parameter since it will remain a static value but the number of connected mobile nodes will vary fairly often over time), R_s, $\bar{q}_s$ and $\frac{1}{\bar{\mu}_s}$ are Exponentially Weighted Moving Averages (EWMAs) of the observed response time, queue-size and service time of s, respectively. The queue-size estimation grows at a cubic rate to punish higher queue sizes. If it grew at a linear rate then it would be possible for a server s_1 with service time n times lower than s_2 to hold a queue n times bigger than s_2 and still have the same expected latency. As explained by the authors of [17] such scenario would be problematic if s_1 suddenly slowed down or even ran into a halt. Finally, the term $os_s \cdot w$ is also known as a concurrency compensation, which is the term that weights the most in the queue-size estimation and it encourages clients to better distribute their requests through the available replicas, effectively achieving a load-balanced system.

The rate control and backpressure module is used in C3 because, according to the authors, replica selection alone cannot ensure that the combined demands of all clients on a single server remain within that server's capacity. On the other hand, the authors of [3] conclude that the complicated rate control mechanism of C3 itself is not helpful to reduce tail latency. Our conclusion is that, although such component might have a positive impact on systems such as the Cassandra Cluster presented for evaluation purposes in [17], where all nodes are continuously engaged in high throughput / high bandwidth operations, for most systems that will not be the case, especially in MEC environments where nodes are power constrained. Also, we already have the *concurrency compensation* baked into the replica selection algorithm to prevent saturating a specific server; And finally, although we understand how this client-side component protects the whole system, we argue that it might not be so interesting and even hurt latencies more than it helps on systems where there might not be many available replicas to start with, because if a low *sending rate* threshold was to be configured we could easily hit the quota on all replicas and incur in unnecessary waiting times imposed by the client itself. For these reasons, we decided to discard the rate control and backpressure component.

4.3.2 Remaining Challenges

With C3 ranking scheme as the basis for our algorithm, we have enough information to understand servers' resources occupation and network conditions, as well as the ability to avoid herd behaviors. We also get intrinsic load-balancing. But there are additional challenges related to MEC environments, namely churn, power efficiency, metrics freshness and a dynamic set of replicas which is not known at start time and evolves over time (nodes join and leave). Also, nodes' computational capacity is linked to the available battery capacity, which drops at a probably non-linear rate over time.

4.3.2.1 Churn

Churn refers to the movement of system nodes. In a cloud environment nodes have a static physical position (usually machines in racks within a data center) and are wired to the network infrastructure which provides highly reliable and high bandwidth connections. On the other hand, MEC systems present a more volatile environment with wireless communication medium that might degrade with physical distance and other environment related variables, as well as mobile nodes which often experiment movement, effectively shifting their physical location. Regarding nodes' movement, there are two possible scenarios:

1. either these nodes move within range of the beacon for the wireless medium (e.g. the AP);

2. or they cross the range boundary, effectively leaving or rejoining the system.

For the first scenario, and because we are using C3 as a basis, we already have what we need to detect nodes' movement. A server's network latency should be directly influenced by their distance to the connected network device. This means that the clients' perceived response time will increase or decrease when the server gets closed or further to the beacon, respectively, and directly impact the computed score.

For the second scenario, however, this might not be of much help. If we consider the scenario where a node leaves the system, the outgoing requests to it will be left hanging. Here we can try to leverage the error conditions. If the application implements some sort of timeouts around the request, we can stop tracking the request upon a timeout and use the configured timeout value as the perceived response time, which penalizes the score as intended. Another more penalizing factor in this case would be the outgoing requests count to that server, which could potentially never decrease; this would greatly increase the chances that the node would not be picked again as the designated replica but, if the node rejoins the system, it would still keep a perpetual penalty. With this, we think it is important for MEC systems to have a request timeout policy in place in order to react and adapt to these situations. As suggested, a timeout would be a great place to adapt client-side metrics. For C3 metrics we can take the chance to register a very high

response time and also discount the timed-out outgoing request. To complement this practice, however, we think we should also have a way to explicitly penalize the score. The most direct way to do this is to inform the algorithm to return only a percentual part of the computed score. This way we could penalize the replica on timeout - or whenever it made sense - by asking the algorithm to, for instance, only consider 80% of the the computed score. We could then raise this back to 100% once we were confident enough that the node was back online.

4.3.2.2 Power Efficiency

Power efficiency refers to how much regard the system gives to the power consumption of power constrained devices. In a Cloud environment, computing nodes are usually plugged to an energy source with no hard limits to its capacity, and therefore this is not a concern. In a MEC environment, however, mobile devices have very specific power restrictions. All mobile nodes are limited by their battery capacity and some have higher capacity than others. It is also likely that nodes will be joining the system will only a percentage of their total battery capacity and that this remaining capacity has to be shared between our application and others. It is of the system's interest to keep the maximum number of nodes online for as long as possible. Thus, our replica selection strategy also has to score replicas according to their remaining battery capacity. Furthermore, it might be interest to observe how a device's battery evolves over time [6, 20]. Devices' batteries drop at different rates and some might even be increasing.

4.3.2.3 Freshness

Freshness refers to how much time has passed since we have registered a given metric value. The longer we hold onto that value the less fresh it will be, or in other words, the less likely it is that it still represents the reality of the system. We argue that the importance of this concept depends on the metric at hand. There is also a set of assumptions we might be able to make depending on the metric.

Server-side metrics tend to be more impacted by freshness as they report a server's state at a given point in time. For instance, a server's reported queue size tells us how many requests it still has to process at a given point in time; the longer it has been since we received that value, the more likely it is it is not true anymore. However, we cannot make any assumption about the evolution of the queue-size; at most we can either be optimistic or pessimistic and assume it has decreased or increased, respectively, or we can try to use past readings to *predict* the curvature of the value. Although these are all valid techniques, they might introduce *entropy* and have a harmful effect to the system. For metrics such as battery percentage, however, it could be possible to make accurate predictions of the current value using past reading to understand at which rate it was decreasing, or even in certain cases, increasing, which is possible if people are carrying powerbanks or have other sources of energy available.

For client-side metrics such as perceived latencies we might not be able to make any assumptions as well. For others, such as outstanding requests, we always know the exact value. Still, there might be systems that introduce other kinds of client-side metrics.

We see freshness as a concern that should be evaluated for each metric individually. With this kind of granularity we can better fine tune our system. Also, we take into account the possibility that, for some systems, metrics' readings might reach the client or be computed in an asynchronous fashion and, therefore, there is not only no guarantee that there is a value for each considered metric at the same time, but it also implies that some metrics might have *fresher* values than others.

There are multiple possibilities to implement this. We could easily define a TTL for each metric and, once that TTL was reached we could disregard the value for that metric. There is also the possibility to define a decay function for each metric, effectively *adapting* its value over time. This last option seems more attractive to us as it allows for a better expression of how a metric reading should evolve over time and it still allows for the implementation of the first option.

4.3.2.4 Unknown Replicas

All the studied Cloud systems consisted of static clusters where, although there could be some process in place for nodes going offline and coming back online, there was no need to worry about new nodes other than those configured initially joining the cluster. In a MEC system, however, nodes are constantly leaving and new nodes joining. Due to this, more often than usual we might be faced with the situation where we have unknown servers within the available replicas list. Here we can take one of several approaches:

- we can be pessimistic and always consider these replicas last;
- we can optimistic and favour them;
- or we can try to be impartial and prefer them over the *bad* replicas but only after the *good* ones.

To define what are *good* or *bad* replicas we need to define a threshold for the scores. Furthermore, it is easier to define said threshold if we first define a closed interval for the possible score values. Given an algorithm that produces replica scores within $[0;x]$, the neutral score value would be $\frac{x}{2}$. We could thus classify the unknown replicas with this score value and effectively achieve the impartial option listed above.

4.3.3 Proposed Algorithm

Having looked at the additional challenges introduced by the MEC environment, we now present our proposed algorithm, Algorithm 1.

First, we will look at the scoring formula. As stated before, our proposed algorithm expands on top of C3. We evaluate a metric on their internal state (queue-size and service

time), on how much demand we already have over that server (outstanding requests) and on the perceived end-to-end latency (response time). We store the outstanding requests indexed to the server and request identifiers in order to easily resolve requests as well as being able to count them. All the other metrics are stored as EWMAs as in the original C3 paper [17]. We also report the battery capacity from the server-side so we know how much battery that replica has left. Having the battery capacity and the expected latency for that replica, we translate these values into the same scale. Now that we have two distinct values of the same order of magnitude (remaining battery and expected latency), we attribute a percentage weight to each one to form the final score value.

Algorithm 1 Replica Selection for MEC Systems

1: **procedure** SCORE(metrics)
2: **if** *hasEssentialMetrics*(*metrics*) **then**
3: *queueSize* ← *getQueueSize*(*metrics*)
4: *serviceTime* ← *getServiceTime*(*metrics*)
5: *outstandingRequests* ← *getOutstandingRequests*(*metrics*)
6: *responseTime* ← *getResponseTime*(*metrics*)
7: *concurrencyCompensation* ← *outstandingRequests* ∗ *CONCURRENCY_WEIGHT*
8: *queueSizeEstimate* ← $(1 + concurrencyCompensation + queueSize)^{QUEUE\_FACTOR}$
9: *expectedLatency* ← (*responseTime* − *serviceTime*) + (*queueSizeEstimate* ∗ *serviceTime*)
10: *latencyScore* ← *MAX_SCORE* − *expectedLatency*
11: *battery* ← *getBattery*(*metrics*)
12: *score* ← *BATTERY_WEIGHT* ∗ *battery* + (1 − *BATTERY_WEIGHT*) ∗ *latencyScore*
13: **return** *score*
14: **end if**
15: **return** *NEUTRAL_SCORE*
16: **end procedure**
17: **procedure** DECAYBATTERYMETRIC(recordBatteryValue, recordTimestamp)
18: $bytesPerMillisecond \leftarrow \frac{totalTransferredBytes}{elapsed\ time\ since\ first\ message}$
19: *metricValueAge* ← *System.currentTime*() − *recordTimestamp*
20: *transferredBytesSinceLastRecord* ← *bytesPerMillisecond* ∗ *metricValueAge*
21: *concurrencyCompensation* ← *transferredBytesSinceLastRecord* ∗ *CONCURRENCY_WEIGHT*
22: *expectedSpentBattery* ← *concurrencyCompensation* ∗ *BATTERY_PER_BYTE*
23: **return** *recordBatteryValue* − *expectedSpentBattery*
24: **end procedure**
25: **procedure** ONTIMEOUT(node, requestId)
26: *requests* ← getRequests(*node*)
27: *requests* ← *requests* \ {*requestId*}
28: *recordResponseTime*(*node*, *TIMEOUT_THRESHOLD*)
29: *penalizeScoreBy*(*PENALIZATION_PERCENTAGE*)
30: **end procedure**

When any metric that is essential to the calculations is missing, or when there are no metrics at all, we fallback to the neutral score, which means that we cannot (yet) classify

that replica according to our parameters. The neutral score is the middlemost value from the closed interval of possible score values. With this we divide the scale in half, where the scores falling on the left side of the neutral score are seen as "less good" and the ones on the right side are seen as "preferable". As explained in the previous section, this is because we do not want to take neither an optimistic nor a pessimistic approach towards the unknown.

Regarding freshness, we consider each metric individually. When considering queue-size, service time or response time, there is no obvious answer as to how these values can evolve over time. We found that simpler solutions such as an always optimistic or pessimistic approach would more than often lead to incorrect predictions. A better approach would be to apply a statistical prediction function to past data points, such as a linear regression, in order to predict the current value. For these to be effective, however, we need a continuous stream of values, and the absence of such stream is why we might need to deal with freshness in the first place. Since we already keep a EWMA for each of these metrics, we decided to skip having a decay function for any of them. The outstanding requests number is an exact number that the client always knows and does not vary over time without it being registered so, it too dismisses the need for a decay function. For the battery capacity, however, we keep track of how many bytes are transferred between the server and the client (upstream and downstream) since we first connected to the server and create a ratio between those bytes and the elapsed time to obtain a *byte per millisecond* rate. We multiply that rate for the last received value's age to have a rough estimation of how many bytes might have been transferred, in average, between that server and any other node. Then we multiply this value by a *concurrency weight* just like in C3 to account for the other nodes in the system and obtain some sort of *concurrency compensation*. And finally, we multiply this value by a system configured rate that represents how much battery a node spends when it sends or receives 1 byte. The produced value from these multiplications represents the battery decay estimation for the elapsed time since the last feedback. We then subtract this value to the current value to obtain the final considered battery value. We also make sure that this subtraction does not go below zero. As discussed in Section 4.3.2.2, there is also the possibility for the battery to increase if the mobile device gets connected to a power source. However, considering the curve of the battery instead of (or complementary to) the absolute value would make our calculations more complex and possibly bias clients to pick that replica. Instead, we decide to degrade the last received reading over time, with an adjustable rate. If a node's battery levels are increasing, we definitely will capture that increase on sequent readings, which will replace the previous ones. If no updates are received, we assume the battery level is degrading. In this scenario, given that we only degrade because we are not receiving metric updates, and metric updates are piggy-backed in all types of traffic, it means we are receiving no traffic from that node; if no traffic is received from a given node for a long interval, the decay penalty will not be very steep either. This ends up being the most simple and elegant solution.

Lastly, when a request to a server times out we adjust the client-side metrics by removing the outstanding request from the list and recording the configured timeout span as the response time. However, because this might not be enough to make the client immediately prefer other replicas, we also apply an explicit penalty. With the explicit penalty we tell the client to only consider a percentage of the computed score as the effective score to compare to other replicas'. This penalty can be reverted by the client whenever it makes sense.

4.4 Integration with Thyme GardenBed

Now that we know how our replica selection module works, we will go over the integration with Thyme Gardenbed [14]. In this section we will talk about Thyme and GardenBed's architectures and then we will show where we have connected the pieces of our module, as well as discussing some of our decisions.

4.4.1 System Architecture

We thus start by describing Thyme GardenBed's architecture. This section is adapted from [21] as it already provides a concise explanation of the whole system architecture. Here we only take the necessary parts to understand the integration process, omitting some more detailed information that is irrelevant for this work and would otherwise overwhelm the reader.

4.4.1.1 Thyme

We present the full architectural picture of Thyme in Fig 4.6. Below we give a brief explanation of the necessary components.

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

Storage. Thyme fuses the Publish/Subscribe infrastructure with the actual storage system by using the virtual nodes to store published content and subscriptions, while cells act as virtual P/S brokers. Upon a notification, the P/S module delegates to the storage the decision to either download the object or ignore it. This is where the replica selection process occurs in the context of downloads.

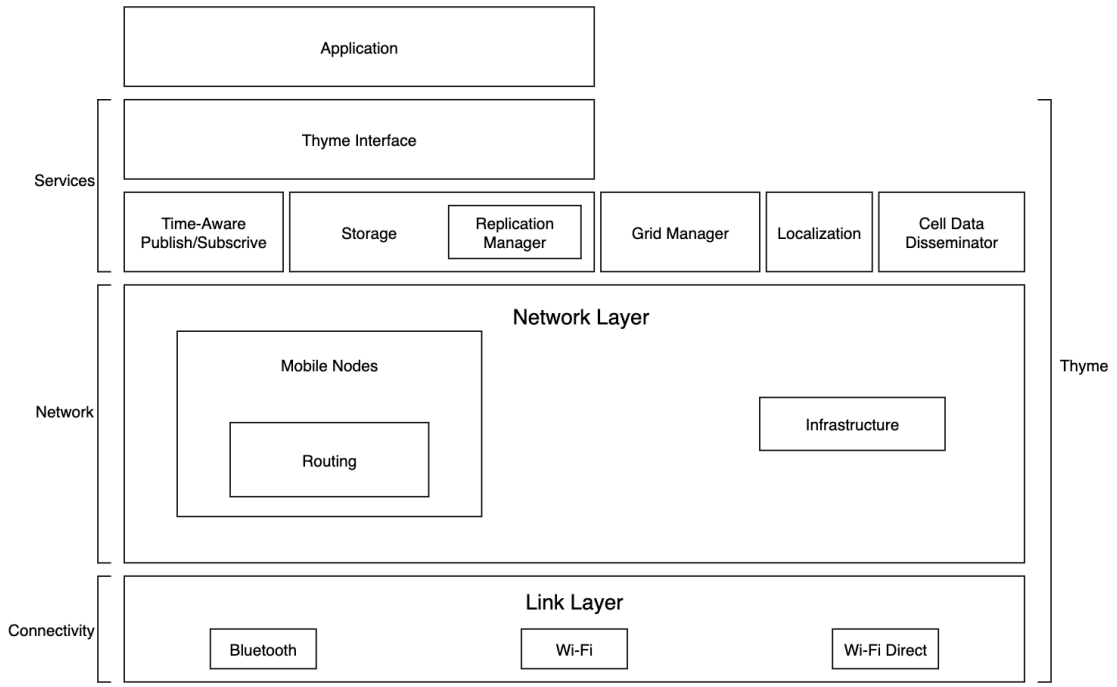


Figure 4.6: Thyme's Architecture

Mobile Nodes Network Layer. This layer allows the communication between mobile devices running Thyme in their front-most application. It offers a custom node identification service that uses logical addresses instead of more classical approaches such as a TCP/IP address for each device.

Routing. The Routing layer was developed with the goal of managing the transmission of messages from one point of the system to another, by choosing the most appropriate dissemination route. It is also this layer that knows how to forward messages to cells/-clusters.

Infrastructure Network Layer. This component enables the communication with the nearest infrastructure, if present.

4.4.1.2 GardenBed

Here the only layer we care about is the **Network Layer**. This is because the supporting MEC infrastructure nodes only act as servers and therefore our only change here is to guarantee we piggy-back feedback metrics on each message, as we will explain in the section ahead. This layer is divided into two parts, the Mobile Nodes and the Infrastructure, and has the goal of allowing not only the infrastructure nodes to communicate

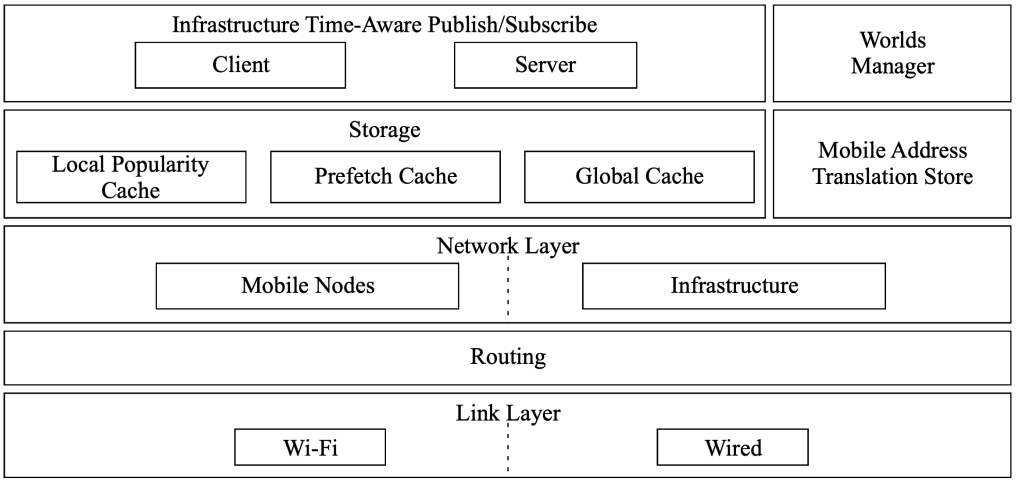


Figure 4.7: GardenBed's Architecture

wirelessly with its mobile clients and through wired links with other base stations, but also to process incoming messages from those sources. Nonetheless, we leave the full architecture diagram in Fig 4.7.

4.4.2 Integration

Now that we have looked within Thyme and GardenBed's architectures, we will discuss the Replica Selection integration.

4.4.2.1 Server-side

First, we needed to disseminate metrics. Metrics are disseminated by servers and each node on this system is a server: it is either a mobile node, which is both a server and a client, or an infrastructure node, which is only a server. Servers piggy-back their metrics on each sent message. We started by adding a *Metrics* payload within the application message. Because we depend on the application's serialization mechanism, and Thyme GardenBed makes use of Protocol Buffers, we also had to add the metrics payload to the message *.proto* definition and edit the marshalling and unmarshalling logic to map between the application and proto generated message formats.

On both Thyme and GardenBed we have created a new *Metrics Aggregator* 4.2.1.2 component which sits on both's *Mobile Nodes Network Layer* and can thus inject the metrics into the message before forwarding it. On both cases we used the *StandardMetricsAggregator* which we provide out-of-the-box and offers the exact expected functionality from the interface.

The final step needed to successfully disseminate metrics is to actually gather said metrics. We need to register *Metric Collectors* 4.2.1.1 into the *Metrics Aggregator* in order for it to bundle some metrics. The metrics we want to gather are: *request queue-size*, *service time* and *remaining battery*.

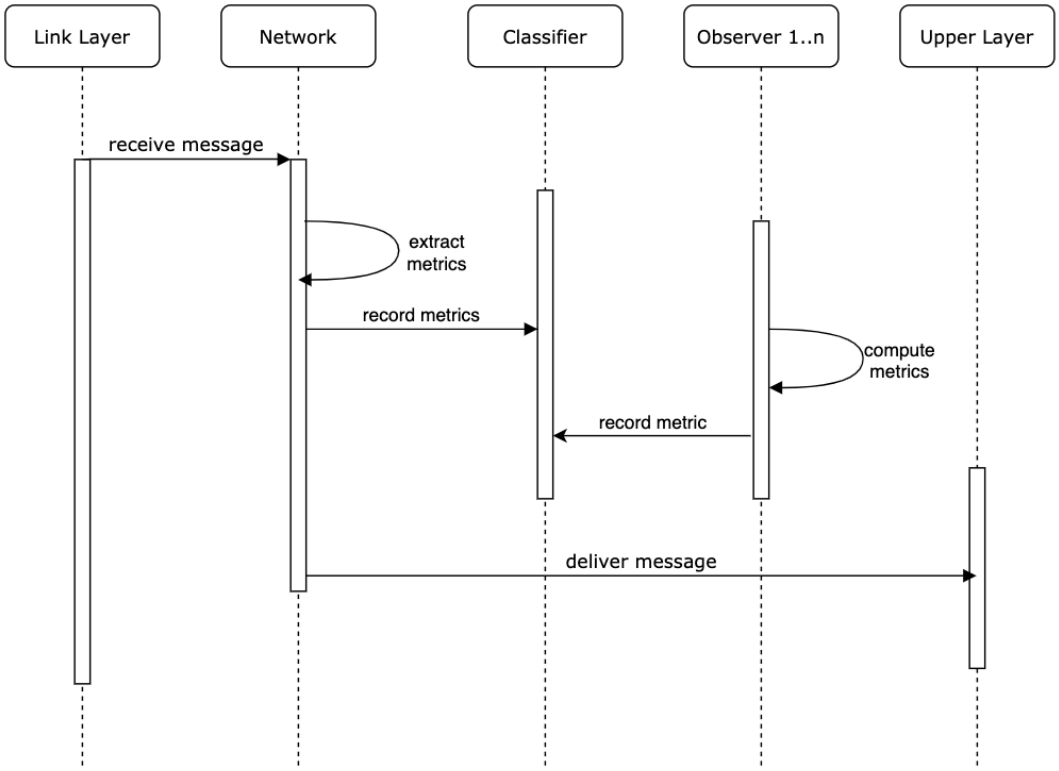


Figure 4.8: Server-side metrics collection sequence

To gather the *request queue-size* in Thyme, we have enhanced both the *Mobile Nodes Network Layer* and the *Infrastructure Network Layer* to expose the number of requests yet to be processed. This functionality consists of an atomic integer that keeps track of the working queue size for each network layer and is exposed to the other layers. We then create a *Metric Collector* that gathers and adds these two counters. The pending mobile requests plus the pending infrastructure requests represent the total number of pending requests in the Network Layer. The same approach is taken in GardenBed as we enhance both the *Mobile Nodes* and *Infrastructure* components in the Network Layer to report their queue-sizes as well. With these we can know all the pending mobile requests to that infrastructure node as well as the inter-infrastructure requests. We then create a similar *Metric Collector* that adds both counters.

To gather the *service time* samples we also had to enhance the Network layer of both systems. We have created hooks within these layers to be able to tap into and receive metric reports, such as the service time with represents how long the system took to process the last request. We then use these hooks to install our *ServiceTimeCollector* which we provide out-of-the-box. This will record the service time samples and will return the average value whenever queried for it. We have also decided to filter the messages we consider here in order to avoid skewed times between mobiles and infra. Thus, and because we decided that we would not specifically differentiate between mobiles and infra, we only consider the common messages, namely the subscriptions and downloads.

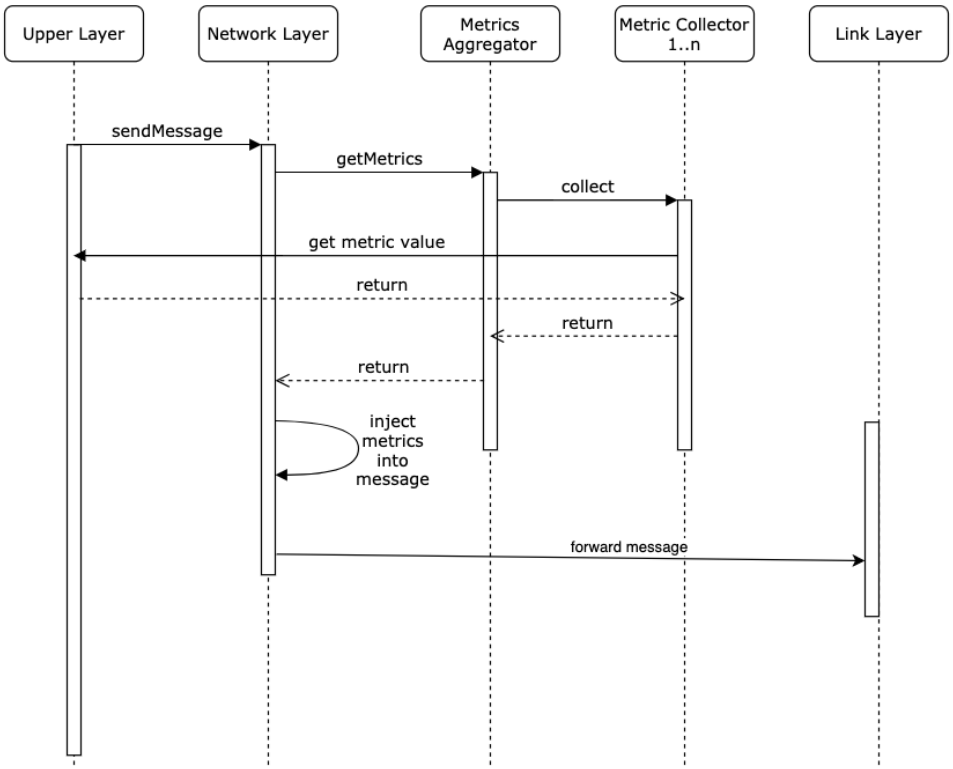


Figure 4.9: Client-side metrics collection and recording sequence

Finally, to gather the *remaining battery* in mobile devices we have created a custom *Metric Collector* that queries the Android battery service for such value. To keep the infrastructure nodes homogeneous with the mobiles, we also report the battery level from those - even though they are not battery constrained - with the maximum capacity. This enables the infrastructure nodes to still be preferred power-efficiency-wise but, with the remaining metrics, they can be considered only after any other available replicas if the expected latencies differentiate much.

4.4.2.2 Client-side

When talking about clients we will only refer to mobile devices running an instance of Thyme. GardenBed servers can also be clients (of other infrastructure servers), but it is out of the scope of our work. Nevertheless, these could also be adapted to use our framework and form a feedback system between them.

To register the metrics and compute scores for each replica we first need our central piece, the *Replica Classifier* 4.2.2.1. Here we use the *StandardReplicaClassifier* we already provide out-of-the-box. The first thing we have to define is the replicas' identifier type. We chose to use the *Address* structure that represents the virtual addresses we have referenced in the previous section. These consist of a UUID and a cell identifier. There is a reserved UUID to represent cells instead of individual nodes, and we have also defined a new reserved UUID and combined it with an otherwise invalid cell identifier, -1, to create

a static address for the infrastructure server. Then we have injected a custom *Cluster Logic* 4.2.2.5 component to be able to identify and correctly score cells, also known as active replicas, which are always the first storage location for the object within the system. We identify a cell by the UUID in the virtual address, and we then are able to retrieve all its known composing nodes by the cell identifier. This will then allow for the classifier to use the average of the nodes scores to score the cluster. This was explained in greater detail on Section 4.2.2.5.

We also register the *Replica Scoring Algorithm* 4.2.2.4 on the bootstrap of this component. This is essentially the scoring formula that we have described in Section 4.3.3. Still on the bootstrap of the *Replica Classifier*, we register the *Metric Holder* 4.2.2.3 class for each metric. We use the *ExponentialMovingAvgMetricHolder* that is provided out-of-the-box to hold the values of the *queue-size*, *service time* and *response time* metrics as EWMAs. For the *remaining battery* we use a simple *RecordMetricHolder* which simply holds the last recorded value; and finally, for the *outstanding requests* we use a *CounterMetricHolder*.

When registering the *Metric Holder* for the *remaining battery*, we also register a decay function to account for the value's age since we might not receive any feedback from that server for a large time window even though it still is within the system and its remaining battery will still decay. This function is as explained in Section 4.3.3. For this we needed to introduce a new component within the Network Layer, which will register a timestamp for the first transferred byte between the client and the considered server, and will also keep a registry of how many bytes where sent and received since. The *Metric Holders* already keep a timestamp for the last value registry and the *concurrency weight* used for the calculations is the same used for the C3 computations. Lastly, the *battery per byte* rate is a number that can be configured within Thyme configurations. These are all the necessary components to compute the decayed battery level. Also, to preserve the correct state of the system, if the value falls below the scale minimum, it is assumed as the minimum, which in this case is zero.

This concludes the bootstrap of the *Replica Classifier* but we still lack a way to capture client-side metrics. As described in Section 4.2.2, we need *Metric Oberservers* 4.2.2.2 in place to be able to keep track of the *outstanding requests* as well as *response times*.

The *Outstanding Requests Observer* is installed on the Network Layer and each time a request is sent to server s, it notifies the *Replica Classifier* to increment the counter on the *CounterMetricHolder*. Similarly, whenever we receive a response from server s, it notifies the *Replica Classifier* to decrement the counter on the *CounterMetricHolder*. Because not all outgoing messages are requests and not all incoming messages are responses, we need to apply message filters to the increment and decrement notifications of the observer, otherwise we would possibly see incorrect values for the *outstanding requests* metric.

The *Response Time Observer* is installed on the Publish/Subscribe and Storage services to keep track of the outgoing requests and incoming responses. Although all these messages have to go through the Network Layer, this upper layer was easier to install such observer as it already tracks requests. When a request is sent, the observer stores the

request identifier alongside the replica identifier and a timestamp. Then, when the request eventually resolves - either with a response or a timeout - we measure the elapsed time and notify the *Replica Classifier* to record that response time sample indexed to the replica.

The last step to complete this feedback system is to include the *Replica Classifier* component in the necessary layers. First, we need to have this component tap into the Network Layer in the same way that the *Metrics Aggregator* does; however, instead of injecting the metrics into the message before it is sent to the Link Layer, now we want to extract the *Metrics* bundle from the payload before we forward the message to the upper layers to be processed. We thus include our *Replica Classifier* on the *Mobile Nodes* and *Infrastructure* components from the Network Layer and have it pre-process all incoming messages to consume the included metrics. With this we have finished the setup and can now make informed decision on which might be the most interesting replica to download a data object from.

Algorithm 2 Previous Replica Selection Policy

```
1: retries ← 0
2: forced ← false
3: procedure processDownload(objectID, replicas)
4:     if isInInfra(objectID) AND retries == 0 then
5:         downloadFromInfra(objectID)
6:     else if retries == MAX_RETRIES then
7:         cell ← getCellAddress(replicas)
8:         download(cell, objectID)
9:         forced ← true
10:    else
11:        replica ← replicas[0]
12:        download(replica, objectID)
13:        if isCell(replica) then
14:            forced ← true
15:        end if
16:    end if
17: end procedure
18: procedure OnTimeout(request)
19:    if retries == MAX_RETRIES OR forced then
20:        fail(request.objectID)
21:    end if
22:    remainingReplicas ← retries == 0 ? request.replicas : request.replicas \
   {request.chosenReplica}
23:    retries ← retries + 1
24:    processDownload(request.objectID, remainingReplicas)
25: end procedure
```

Now we can sort the available replicas for a given download. To do so, we need to include the *Replica Classifier* in the Storage Layer which, as stated before, is responsible to handle a *Notification*. The *Notification* data structure includes the metadata for the data

object, including the available replicas to serve the object and if a copy of the object exists in the infrastructure. The Storage layer already had a replica selection policy which is synthesised in Algorithm 2. The strategy was to **always** prefer the infrastructure first in order to conserve the other devices battery as much as possible. If the infrastructure had already been tried or was never an option, we started visiting the replicas by the order they appear in the *Metadata*. We would try at most as many *passive* replicas as the possible number of retries before being forced into downloading from the *active* replica (the cell). If we ever found the cell in the ordering before running out of retries, it would still be the last try since it was assumed that if the object did not exist in the active replica then it had been removed from the system. Thus, if that last resort failed, we failed the download. The changes we have made in this policy were that, first we are not immediately considering the infrastructure in the first place anymore; it can still be the first option, but it is not guaranteed that it will be. Before starting to process the download, we check if the object is indeed available in the infrastructure. If it is, we add the reserved infrastructure *Address* to the metadata replicas list. Then, we use the *Replica Classifier* to **sort** the metadata replicas list and only after start processing the download. We still keep visiting the replicas by their order but now they are sorted by our criteria. Furthermore, we keep the timeouts, retries and the forcing of the active replica if we ran out of options. We can see the revised policy in Algorithm 3.

With this, we finish our integration of our Replica Selection framework and Algorithm 1 with Thyme GardenBed.

4.4.3 Dealing with Early Hotspots

At this point, there is still one problem that remains unsolved. Inspired by Figure 3.2, we have proposed the following hypothetical scenario: We imagine a football match where the people in the stadium are using a multimedia sharing application supported by Thyme. In the beginning of the game, a big part of the users make a future subscription to the tag *goal*, spanning the expected duration of the game. Then, at some point during the game, there is a goal and everyone starts publishing photos and videos to the tag. What follows is a barrage of notifications from the broker cell to the possibly interested devices. Then, most of these devices will potentially try to download each of these files at the same time, resulting in a really high demand over the publishers' cell.

In this scenario, there is not much a replica selection algorithm can do because there are no alternative replicas to select from. The only nodes holding these files are the publishers and eventually their cell peers, meaning that only the active replica will be listed as the available replica in all those subscription match notifications. Because of this, the cell will be serving all the downloads resulting from a notification for a file it owns. This, in turn, results in a steep battery capacity reduction on the cell nodes, which might ultimately cause them to leave the system. Moreover, because bandwidth and computing power are specially limited, nodes might start to queue up download requests, causing

Algorithm 3 New Replica Selection Policy

```
1: retries ← 0
2: forced ← false
3: procedure BEFORESTARTINGTOPROCESSDOWNLOAD(metadata)
4:     if isInInfra(metadata.objectID) then
5:         metadata.replicas ← metadata.replicas + {INFRA_ADDRESS}
6:     end if
7:     sort(metadata.replicas)
8:     processDownload(metadata.objectID, metadata.replicas)
9: end procedure
10: procedure PROCESSDOWNLOAD(objectID, replicas)
11:     if retries == MAX_RETRIES then
12:         cell ← getCellAddress(replicas)
13:         download(cell, objectID)
14:         forced ← true
15:     else
16:         replica ← replicas[0]
17:         download(replica, objectID)
18:         if isCell(replica) then
19:             forced ← true
20:         end if
21:     end if
22: end procedure
23: procedure ONTIMEOUT(request)
24:     if retries == MAX_RETRIES OR forced then
25:         fail(request.objectID)
26:     end if
27:     remainingReplicas ← request.replicas \ {request.chosenReplica}
28:     retries ← retries + 1
29:     processDownload(request.objectID, remainingReplicas)
30: end procedure
```

them to timeout.

From our definition of *"popular"*, we understand that these files are already popular, or at least, have the potential to be. However, by the system definition, *popularity* is a product of a file's demand, or in other words, how much they are downloaded. In the presented scenario, it would be ideal if these files could be served from the infrastructure right from the beginning; but picking the infrastructure as the target download location is a concern of the downloading node, and it will not target the infrastructure because the initial notification states that only the active cell contains the file. Therefore, we need a way to better preserve the replicating cell without depriving nodes from the requested files.

One first solution that came up was to delay notifications, or send notifications in batches. Effectively, if the broker cell did not send all notifications at once, but instead waited until the first notified nodes downloaded the object and only after notified another

batch, now with more storage locations, the active replication cell would not be so severely punished. However, this does not seem like a really good solution because it intentionally degrades the quality of service for some nodes in favor of others.

Another possible solution, and the one we have picked, is to redirect the download requests to new replicas. For this, we have introduced a new configurable setting into the system: the threshold value of concurrent downloads one node can serve. All the downloads beyond this threshold should be redirected to another replica whenever possible. To redirect a download to another replica, we have leveraged the existing *Download Fail* message, adding a redirection flag. We have also added a section to include the most up-to-date metadata for the target object, which will include the new storage locations. Instead of a *File Message*, the serving node sends a *Download Fail* message in response to the download request, containing these extra fields. This gives the downloading node the possibility to pick a new replica to fetch the file from, now using proper replica selection.

4.5 Final Remarks

In this chapter we started by presenting the framework we have created to be the basis of our solution for replica selection in MEC systems. We made a deep dive into each component and saw how the programmer could use them to build a replica selection mechanism into his system. Then, we went over our proposed algorithm and how it deals with the extra challenges introduced by the volatility found in MEC environments. And lastly, we analyzed the integration of our framework and algorithm with the existing MEC system, Thyme GardenBed.

Now that we have a solution in place we need to assess how effective it is. To this end, we have designed and ran some scenarios to answer some fundamental questions regarding the validity of our solution. Next chapter goes over the evaluation process and presents our experimental results.

Chapter 5

Evaluation

In this chapter we will go over the experimental evaluation process we used to validate our framework and its integration with Thyme GardenBed. We will first discuss which questions do we want to answer with this evaluation, then we will explain how we can answer those questions, followed by the necessary experimental setup to apply our methodology. Having discussed all the preparations, we will then discuss the experimental results, if they were expected and what did we learn from them. Lastly, we close this chapter by summing up our findings.

5.1 Goals

Before being able to evaluate anything, we have to decide **what** to evaluate. Since we have built a replica selection mechanism and had it integrated within an existing system, the big question is *how much did the system improve*? In the broad sense, and as explained before, our aim was to improve latencies and overall system resources usage (such as battery capacity), as well as being able to better react to network congestion and system's performance degradation. In Chapter 4, Sections 4.3.1 and 4.3.2, we have explained in detail the faced challenge and in Section 4.3.3 we have explained how these principles are baked into our work.

Because these concerns are properties (or desired side-effects) of our solution, it would not make sense to test each of them separately. Thus, we gather them all into a simpler question: *how good is the replica selection process?* In other words, do we consistently pick the best replica amongst the available ones? Ideally, we should be able to compare our algorithm's decisions with which would actually had been the best option available option. With this, we could compute an average error value that represents how accurate our replica selection strategy is. Furthermore, we could apply the same methodology to the Thyme's previous replica selection scheme as well as other baselines that might be interesting to consider.

Even with the best strategy, the replica selection process can only be as good as the available information about the replicas. Little information can lead to bad, uninformed

and even arbitrary decisions. Stale information, on the other hand, can lead to confident bad decisions. Ideally, we would know about any system change as soon as it happens; however, that is usually (if not always) not the case. Thus, it poses the question: how fast can we perceive this changes? How quickly can we propagate and perceive this information? *How reactive is our system?*

Lastly, we would like to know the overhead introduced by our module. Effectively, we are introducing more bytes into the network by piggybacking system metrics on existing messages. Thus, we would like to know *how much*. In short, we want to know what is the increase in traffic volume and how does it relate to performance gains.

In summary, we want answers to the following questions:

- How good does our replica selection strategy perform?
- How reactive is our solution?
- How much overhead does it introduce and how does it relate to the results for the first question?

5.2 Methodology

Now that we know **what** to evaluate, we are missing **how**. In this section, we discuss our experimental methodology and how the produced results answer the questions from Section 5.1.

To assess whether or not we consistently pick the best replica, we need to know two things for each download: the chosen replica and the actual best replica. To know which replica was chosen is just a matter of recording it; to know which would have otherwise been the best choice, however, requires a bit more effort. To know the latter, we came up with the idea of an *oracle*.

The *oracle* is an extra piece of software that needs to be fed with the whole system information to be able to answer any question with the highest degree of certainty. It is composed of two parts:

- an extra persistent logging component that is enabled within the system nodes to record their state and downloads information;
- a post-processing script which computes metrics (e.g. how good was the replica selection on each specific download) over the previously collected runtime data.

With this we can now answer our first question, "*How good in our replica selection strategy?*". To start, each node creates a new record for each download, containing the important information that would impact the replica selection decision, such as the available information about the system (i.e. which metrics does it have for each of the nodes it knows at that point in time), as well as the replica it chose. Furthermore, each nodes records

a snapshot of itself at each second, containing its most up-to-date metrics and stored (replicated) objects.

Using the download records, we can compute the degree of effectiveness of each algorithm, i.e., we can assess whether the given algorithm is consistently picking the replicas which best satisfy our previously established concerns by comparing its results to the *oracle*'s and if not, how close it was. We can determine this by comparing the replicas' order as sorted by the node to the order as sorted by the *oracle*. The idea is to run this test in the exact same conditions for different baselines and use the results as a base of comparison. Moreover, we use these results to understand the cost/benefit of more complex solutions - such as ours - compared to simpler solutions such as a random selection algorithm, or other intermediary solutions such as the C3 algorithm designed for Cloud environments and a slightly tuned version for MEC environments.

Using the recorded system snapshots, which contains the most up-to-date metrics of each node and their stored contents at each moment, we can use the *oracle* to compute the optimal replica system-wide to the node's decision. The difference from the previous measurement is that now we are not evaluating whether the selection scheme made the best decision with the information it had available at that moment - keep in mind that even though some algorithms do not use any of our metrics or very few, we assume these are available to all baselines regardless of utilization - but instead we are considering the most up-to-date system snapshot, containing not only the replicas presented in the download notification but also replicas that have not yet been registered as being replicating the target object, or subtracting any node that might have stopped replicating that object; all the nodes' metrics considered for the computations are also the most up-to-date to the download's moment as reported by the nodes themselves. By being able to sort the whole replicas in the system using their exact most up-to-date metrics, we are able to determine just how good a replica selection was system-wide using only the available information in the node. This can give us a rough estimation of how effective the whole replica selection module is, as well as a rough estimation of the *reactivity* of the system. For this test, we defined the following baselines:

1. **Random Selection** - The client sorts the available replica in an arbitrary fashion;
2. **Infrastructure First** - The client always picks the infrastructure whenever it is available;
3. **C3** - The cloud envisioned implementation of C3;
4. **MEC C3** - A slight variation of C3 which is more adjusted to MEC environments;
5. **Wasabi** - Our replica selection algorithm.

Still on the topic of *reactivity*, we want to have a better estimation of how the system reacts to change, specially abrupt variations on replica metrics. Our concerns here are in regards to whether the system re-prioritizes replicas upon one of the preferred ones

becoming a bad choice or even if nodes can get access to such information before it needs to be put at use (i.e. before trying to perform a bad download). To this end, we can create a controlled environment where we use the same set of replicas for consecutive downloads. Then, we can abruptly drop the battery levels of the preferred one(s) which should cause it to rank worst, and wait a variable amount of time before attempting a new download. With this, we can see if the downloading node(s) can get the battery metric updates in time to impact their decision.

And last but not least, we want to measure the overhead of our solution in the integrating system. Specifically, we want to know how the volume of data in transit increases with the piggybacked metrics. Currently, the nodes already report the amount of bytes they send over the network. We can aggregate these measurements of each node to know how many bytes have been in transit for a given test. If we do this for an instance of the system without our module, and one with it integrated, we can then compute a ratio of the total bytes transferred to have a value that represents the increase. After, we can compare this ratio with perceived increase in the quality of the replica selection process and compare the two to understand if there is a good cost/benefit to our solution.

5.3 Experimental Setup

In this section we discuss our testing environment. Each Thyme node represents a mobile device in the topology of the system. However, we do not have access to a sufficient amount of devices to properly set up a real testing environment. Moreover, even if we did, it would be troublesome to constantly update the application and further manage each device. Thus, we resort to a simulated environment which allows us to better control each device and its performed operations, as well as system topology (i.e. how many nodes to use).

5.3.1 Simulator

The simulator we used is one developer in-house and which utilizes a trace-based simulation framework. We represent our simulated environment with a trace file which lists the operations we want each node to perform and when, such as going online or offline, subscribing a tag, publishing to a tag, and so on. The simulation itself is a single process in a single machine, emulating each mobile device in a separate thread (and from here, each device can use as many threads as necessary).

Furthermore, the simulator replaces Thyme's Network Layer to support logical dissemination of messages between any number of virtual nodes. Although the communication between mobile nodes is now captured in a logical layer that removes the need for the Link Layer for mobile-to-mobile communication, the communication between mobile nodes and infrastructure still uses the usual network layer. This is because we do not simulate the infrastructure. Instead, we run the actual infrastructure server, GardenBed, in a

different process and use socket communication with the usual TCP/IP stack to exchange messages.

Each operation within the trace will map to an *action*. An action is the logical representation of the operation. Each kind of action has a mapped *behavior* which represents the effect the action should cause in the system, and the action itself contains the values to parameterize such behavior. Furthermore, we have to provide mock implementations for the runtime components provided by the Android SDK, which let us use and read information about the underlying hardware, such as battery levels. To control the initial battery capacity, we have edited the existing *NODE* action, which spawns a new mobile device, to contain the initial battery value.

5.3.2 Traces

Now that we have discussed the simulator, it is time to discuss our testing scenarios.

We started by choosing a scenario to evaluate how good the replica selection algorithm is. To properly evaluate this, we decided to create a scenario with lots of subscriptions and publications. We also wanted a sufficiently large amount of nodes to be able to span multiple cells. To this end, we have created a script that generates a scenario according to some specific parameters, namely the number of mobile nodes to spawn and the number of tags to use for publications and subscriptions. After testing several scenarios, we decided to go with 64 mobile nodes and 8 distinct tags. Also, as said before, the infrastructure server is not simulated; Instead, we spawn it in a different process. The virtual devices are configured to find the infrastructure server through broadcast discovery, which will allow them to then know the infrastructure address to further open TCP/IP connections.

The generated trace is divided in 4 parts, each representing a phase of our simulated environment:

1. Spawn all the 64 nodes and let them join the system. Only after all the nodes are online and ready do we proceed;

2. All nodes have a 50% change of subscribing to each of the available tags. This will cause some tags to be more subscribed than others, which will cause some published items to be popular later on. All these subscriptions are spanning into the future, meaning that from that point onward, each time some node publishes to one of the subscribed tags, the subscribed devices will all the a notification. We have configured all devices to download the object upon a notification, in order to maximize the number of downloads in the simulation and thus have a more accurate measure of the selection process;

3. After all the future subscriptions comes a barrage of publications. As before, each node has a 50% chance of making a publication on each tag. This will trigger several notifications on each publication, which in turn will trigger the same number of downloads. The most downloaded items might eventually be pushed into the

infrastructure. Because these are all new publications, they will only be available from the active replica, which means that the cell address will be the only one available in the list of replicas. This is not very interesting because there is only one option and thus we are not able to make any kind of selection. However, everything up to this point serves to the purpose of warming-up the system, from having nodes contact each other and exchange metrics to giving the possibility to have the infrastructure as a replica option for the most popular items;

4. In this final step, which spans through most of the simulation, all nodes have a 70% chance to subscribe to the tags they have not subscribed before. These subscriptions, however, are spanning to the past, meaning that each of them will pick all the items published to the same tag on the previous step. In this phase we will continuously have a high volume of downloads in the system which means that smart replica selection might play a big part in load balancing and resource management. Most of the items here will become popular, meaning that as we progress through the trace, more and more downloads will consider the infrastructure as a possibility as well.

We use the same trace to run a simulation for each of the baselines outlined in Section 5.2. Then we post-process the log files from Thyme and GardenBed using our *oracle* to retrieve the necessary data and apply the detailed methodology of Section 5.2.

After these, a different trace is used to test system reactivity. We create a trace with 10 nodes and each makes a future subscription to the same 10 tags. Then, we make each one of them publish an item for the corresponding tag (e.g. node1 publishes to topic1, node2 to topic2, and so on). Then, we introduce an extra node and start a warm up phase to let the nodes *know* each other. Finally, the extra node starts making subscriptions into the past to each of the tags and, between each subscription, we abruptly drop the battery levels of one of the nodes and wait a configured amount of time to perform the next subscription. Specifically, we subscribe to topic 1, then drop the battery of node 1 and wait 9 seconds before subscribing to topic 2 to see if the node perceived the battery change of node1. We repeat this process with all nodes and tags, ultimately reaching the 1 second interval between the last two past subscription. The idea is that, since the available replicas will always be the same 10 nodes, we should compute different orderings of the same replicas when performing the next download - which will happen in response to the notification we receive after subscribing to the past - since we are forcing big battery drops between each download, which should reflect on the affected replica' score. If such reordering happens, we know that the metrics update reached the downloading node on time; if not, we know that the replica selection was made using stale information. there are some other changes we node to make to the system for this trace execution to work as expect, namely, we have to force the client to expand the virtual node address into all the nodes it knows with the same address group. This is because each node is part of the

same cell, and thus the cell address is what is presented as the available replica, not the actual nodes.

We also introduce a third trace which is a smaller variation of the first, i.e., with less nodes and less operations. We opted for 16 ran the same script to generate the trace. The idea is to run this trace twice, with and without our solution. From both simulation instances we can then compute the total amount of bytes transmitted during system execution, and finally compute a ratio between the two values to know the total % increase in bytes caused by our solution. We can also check the % performance different between our solution and the previous one - Infrastructure First - from the first test, and finally compare these two values to have an indicator for cost/benefit.

5.3.3 Hardware

To support our simulated scenario, we made use of the computational nodes cluster available from our college department. We have run the simulator in a AMD EPYC 7281 cpu with 16 cores and 32 threads. The system also had 128 GiB DDR4 2666 MHz of RAM and a 2 x 10 Gbps.

We have run GardenBed in a second node of the cluster with the same specification. The communication between simulator and infrastructure was made through the network connection link while the communication between nodes was made through the logical layer provided by the simulator.

5.4 Results

In this section we present and interpret the experimental results of our system. We split the results into sub-sections, one for each of the questions laid out in Section 5.1, which we answer on the respective section.

5.4.1 Replica Selection Quality

As described in Section 5.2, we have established some baselines to understand how they compare to our solution and how much it impacts the integrating system, Thyme GardenBed. We sorted each baseline by their implementation complexity, starting from a totally arbitrary selection, to always giving preference to the infrastructure when it is available (which was the previously established solution), to making use of feedback metrics with the C3 algorithm, followed by a new version of C3 tuned for MEC environments and finally our solution, Wasabi.

Below we present the same kind of chart for each baseline: the horizontal axis representing time, more precisely instants from the trace execution, and the vertical axis representing, as a percentage, the quality of the replica selection: 0% meaning that the worst possible choice was made and 100% meaning the best. All the charts have two plots: blue and orange. The blue plot contains the points representing the quality of the

replica selection for a given download happening at a certain instant, and exclusively considering the replicas that are listed as copies of the target object within the download notification. The orange plot contains the points representing the quality of replica selection for the same download at the same instant, but now considering a global snapshot of the system where we know the exact metrics for each node at that same moment, as well as what are the actual replicas for the target object. These evaluation metrics have been discussed in Section 5.2. For instance, if some other node has the object in its storage already but is still not in the available replicas list, we still consider it. This is because we want to understand the real impact each replica selection has on Thyme GardenBed and how much room there is to improve the feedback system. Even if a node always makes the best choices with the available information, that information will most likely not be 100% accurate with the current state of the system. These can also help in understanding the *reactivity* of the system. This means that for each download in the chart there will be two points: one in the blue plot and another one in the orange plot.

Moreover, we present additional information regarding how many of the existing replicas were included in the download notification (i.e., how many replicas of that object did the system already know), as well as how good is the score of the selected replica versus the best available option. We compare the computed score for the selected replica to the score of the actual best option amongst the available ones because we want to have one more way to compare each baseline to our solution other than just the ordering of the replicas. Even if a given baseline decides on the second or third replica, it might happen that their scores are very similar and therefore the difference in the selection might be negligible.

5.4.1.1 Random Selection

The Random Selection strategy takes a completely uninformed approach towards replica ranking as it simply shuffles the available list. For this reason, our expectation towards its benchmark was that it would be the worst performer. And indeed, Figure 5.1 proves that the quality of the selected replicas is fairly inconsistent. Looking at the blue plot, even though it takes the best available replica for some of the downloads, these were all scenarios where the infrastructure was an option and, by chance, it was picked. When we say by chance though, the odds where actually favourable. Looking at Figure 5.2 we can see that most downloads are only considering a tiny part of all the existing replicas for that object. In fact, for this simulation we knew **on average** about 40% of the existing replicas on each download. The average number of replicas available on each decision was around 3.6 replicas. Since we were only considering for statistics the downloads with 3 or more replicas, this absolute number shows that most downloads had only 3 replicas to pick from. That is a 33% change to pick the infrastructure if it is amongst the available options. Other than those scenarios, we rarely see any download with a selection score over 80% and there are more below 40% than on any other baseline.

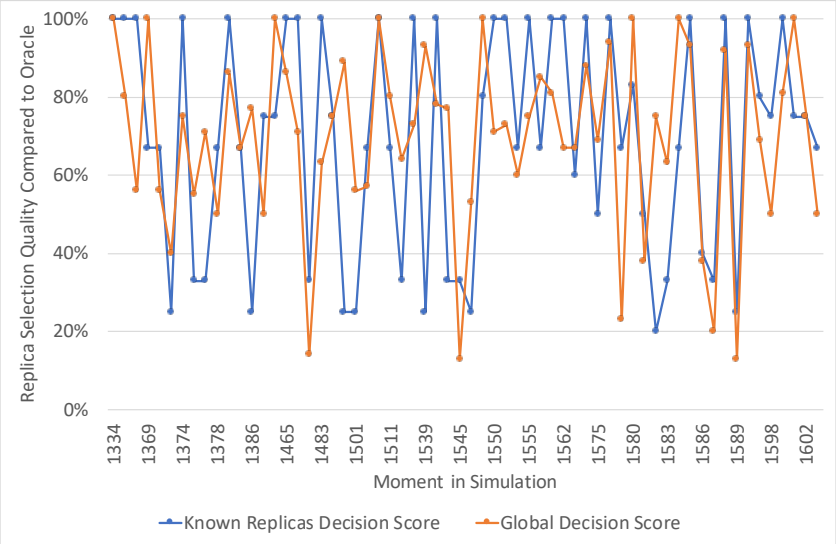


Figure 5.1: Replica Selection Benchmark for the Random Selection Strategy

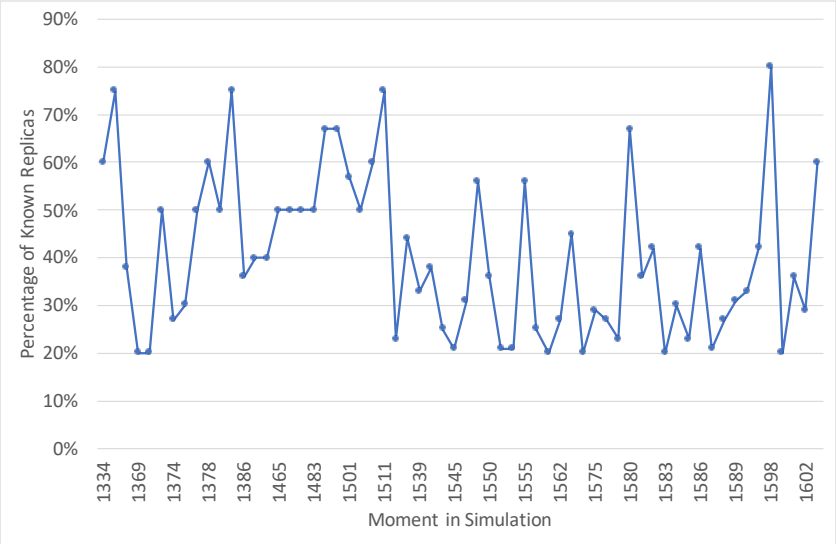


Figure 5.2: Percentage of replicas available in the download notification comparing to the actual number of nodes that already has the object in its storage.

Looking at the orange plot in Figure 5.1, we can see that the quality of the replica selections from the perspective of the *oracle* is not much different. In fact, on average, this strategy scores 69% for replica selection quality over the known replicas and 68% looking at the exact snapshot of the system at that moment. It could be argued that

considering more replicas than on the blue plot, the replica selection quality should be even worst. If we look closely, there are choices which score goes below 20% here, which means that some choices had a big potential to slow down the system. Still, there are some instances where the choice was the best possible one, most of these still considering the infrastructure. It is important to notice that in only very few occasion does the the select replica represent both the best option amongst the ones available for the download and amongst all the existing replicas. When the choice represents one of these maximums, it is usually in an exclusive way. This also hints to the asynchrony between stored metrics and system state. This, however, is something that has no impact on this strategy given that it has no logical selection criteria.

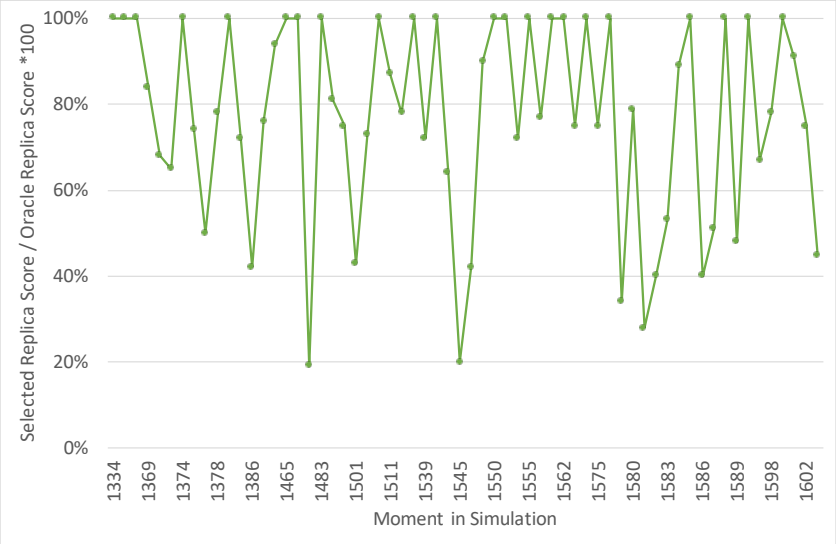


Figure 5.3: Ratio between the select replica score and the actual best replica score for the given download.

Regarding the score difference between the picked replica and the actual best, it also has the steepest curve amongst all the baselines (Figure 5.3), holding on average 78% of the best score. This means that there is enough room to make bad decisions that might slow down the system. Granted, the trace executions for the Random Selection always yielded less total downloads than any of the others, meaning that some download requests incurred in big waiting queues.

5.4.1.2 Infrastructure First

Infrastructure-first was the previous solution and possibly the most interesting one to compare our solution to. The infrastructure is always preferred when it is available, regardless of load and network conditions. We argue that this creates a major possibility

for a *hotspot*. When the infrastructure is not available though, it falls back to the order of the list presented in the download replicas, which should compare to the *Random Selection* since there is also no criteria on that ordering.

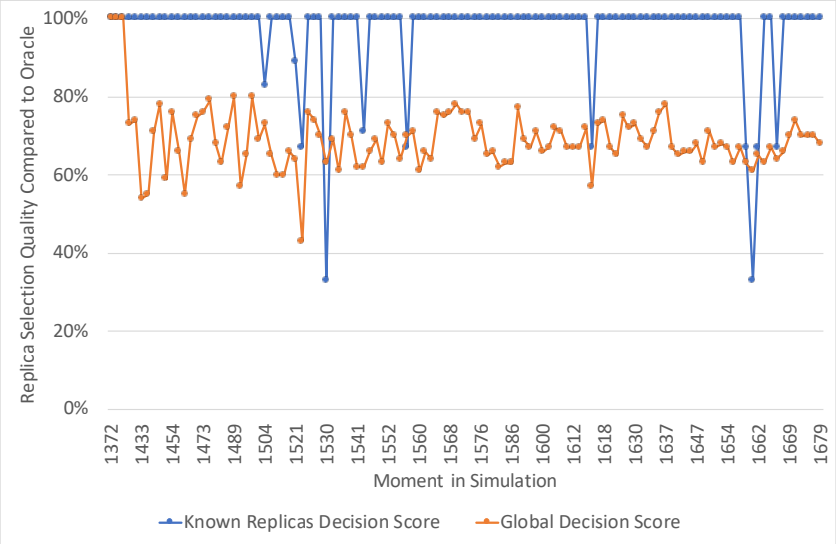


Figure 5.4: Replica Selection Benchmark for the Infrastructure First Strategy

Figure 5.4 shows a much more consistent replica selection quality towards the benchmark, compared to Random Selection. The great majority of downloads were indeed targeted towards the infrastructure and, to some of our surprise, the *oracle* agreed on the infrastructure almost always being the best available option, as depicted from all the 100%'s. After further analysis, we have found that this result was due to several points:

- First, our benchmark was pretty linear and without any artificial network congestion or system pauses introduced. Because the execution environment was a very controlled one, these phenomenon did not occur by themselves. Thus, mobile nodes never experienced large latencies from the infrastructure;
- The infrastructure always reports 100% battery capacity, as it is not a battery constrained server. As previously explained, this is a design decision to make the infrastructure server seamless with the mobile ones, although creating a conservative bias towards it. Because no other device in the simulation started with 100% battery and, at the moment these downloads occurred the batteries had been slightly drained already, contributing to the infrastructure bias;
- Moreover, the infrastructure consistently reported service times in the order of 1 millisecond whereas the mobile nodes had, on average, 132 milliseconds of service time. This can be justified by the fact that we had a dedicated machine for the

GardenBed server, whereas, in spite of the available hardware resources, the mobile devices were all being simulated from the same process and thus had some limitations such as the number of threads a single process can spawn;

- Lastly, we verified that each node was very limited in the number of concurrent requests it could make. Adding this to the time interval between each trace action, the fact that nodes process the simulation trace in a synchronized fashion (nodes have to wait for the others to finish processing the current action before it can move to the next one) and the service time of GardenBed, means that the number of outstanding requests towards the infrastructure would rarely grow. Since this is the most penalizing metric towards expected latencies, the expected latency for the infrastructure would always be reasonable.

To sum up, according to the previous points the infrastructure is greatly favoured in these benchmarks since it reports maximum battery levels at all times and the expected latencies are very reasonable. However, looking at the actual download conditions (orange plot) we can see that the infrastructure was rarely the best option. Still, selecting it always resulted in a selection quality above 60%.

When the infrastructure was not available, however, this strategy falls back to an arbitrary selection, which results in skewed results compared to the alternative. Most of the downloads for non-popular objects have much lower replica selection quality, some even dropping below 40%.

The difference between the selection quality of the available replicas and known metrics versus the actual system state stems mostly from the *outstanding requests* metric, as explained above. Although each node individually was not sending many concurrent requests to the infrastructure, the infrastructure was still receiving many requests at the same time, from different nodes. Thus, each node individually expects a better infrastructure latency than it can actually provide when considering all system nodes. On average, the selections had a 96% quality considering the available information and only 68% over the actual system state. This last percentage is, in fact, very similar to Random Selection's, meaning that this baseline has the same potential for decisions that might slowdown the system.

To compare this scenario with a more download-intensive one, we removed the concurrency limit of each individual node on the simulator and ran the same trace. Figure 5.5 shows a very different story from the previous benchmark. Raising the number of concurrent requests per node, and therefore the number of outstanding requests towards the infrastructure, makes the quality of the replica selection drop significantly when prioritizing the infrastructure. Here, the quality of replica selection considering the available information and the global system state dropped to 73% and 53%, respectively. This tells us that this solution has even more potential for bad decisions than the previous Random Selection (!). This is because it is intentionally creating a hotspot which, in the face of infrastructure delays or network congestion can bring the system to a halt.

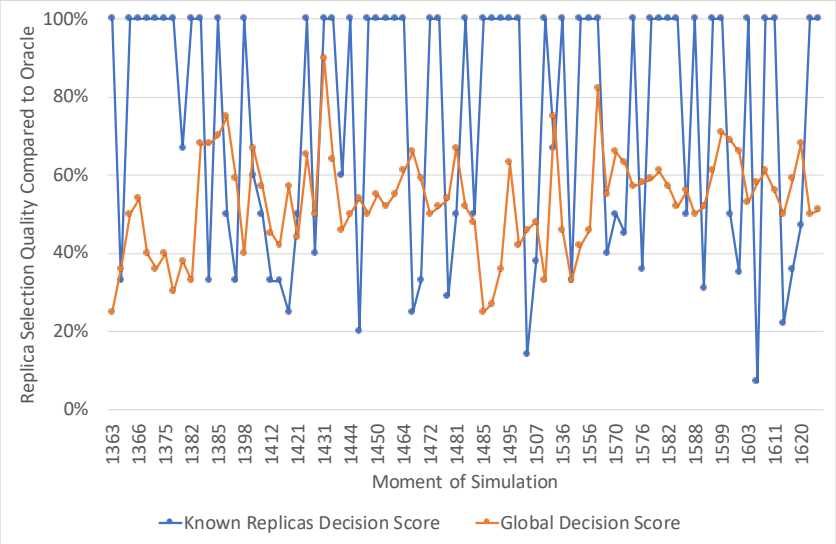
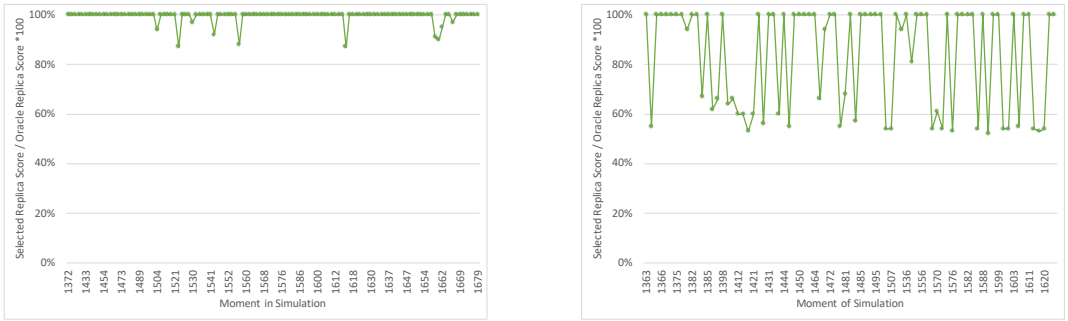


Figure 5.5: Replica Selection Benchmark with extra allowed concurrency for the Infrastructure First Strategy.

Regarding the percentage of known replicas at download time versus the actual number of replicas in the system for the given object, it follows the same trend of the Random Selection benchmark where, on average, nodes had 40% of the existing replicas available in the download notification. This will keep being a trend throughout all benchmarks and we discuss it in more detail at the end of this section.



(a) Benchmark 1

(b) Benchmark 2

Figure 5.6: Ratio between the select replica score and the actual best replica score for the given download.

Finally, regarding absolute score differences, we have two different indicators. On one hand, looking at the first benchmark (Figure 5.6a) we see that the scores of the replicas selected by this strategy were very identical to the ones which would have been picked by

Wasabi, with an average of 99.3% score similarity. On the other hand, when we introduce extra individual request concurrency, this value drops significantly to an average of 83% (Figure 5.6b), following the trend of the chart in Figure 5.5.

5.4.1.3 C3

C3 was our highlighted replica selection algorithm for cloud environments. Since it was the starting point to our work, we included it as a baseline to understand how it performs without being adapted to MEC environments. Figure 5.7 shows that C3 still had an interesting behavior in a MEC environment. Most choices made using the available information were spot on, despite having some distinctively bad ones as well. The average selection quality rate was 93%. The worst choices, mainly the ones we see below 40% were due to the fact that C3 does not account for battery consumption thus, replicas that might soon leave the system due to a power shortage are still selected in order to minimize latencies. Furthermore, C3 does not account for *unknown* replicas, meaning that when there is a replica amongst the available set to which we have no metrics, it is considered only as a last resort. Thus, C3 will pick replicas it knows will perform badly over replicas it knows nothing about.

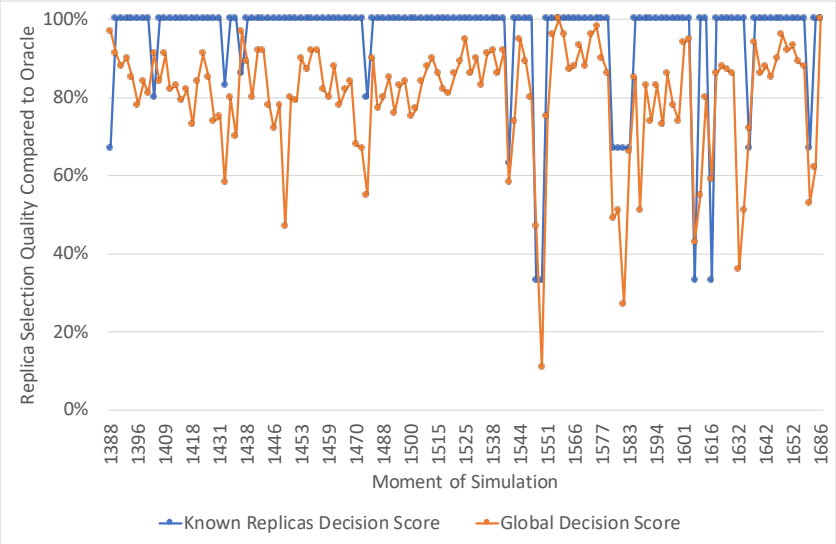


Figure 5.7: Replica Selection Benchmark for C3 in a MEC environment.

Regarding the quality of the replica selections according to the global system state, we see a positive increase compared to the previous baselines, scoring an average of 80%. This is the first baseline to make use of available feedback metrics which means that having a feedback loop between servers and clients can have a positive impact on a replicated system. As we can see on the chart, the worst system-wide selections also

match the worst selections made with the available information. Most selections are still above 80% and, following the trend from the previous baselines' benchmarks, only about 40% of the existing replicas were available for selection on each download. This means that the stored replicas metrics on each client were very identical to the actual system state when performing a download and the gap between both plots can be justified by:

1. The disregard for the battery levels;
2. Considering bad replicas before unknown replicas;
3. Not knowing the best existing replicas on some of the downloads, making it impossible to select them despite selection scheme.

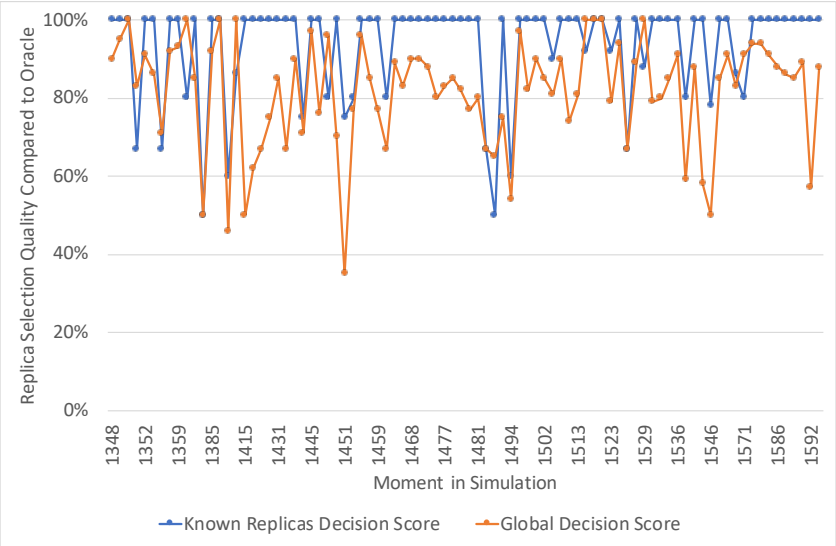


Figure 5.8: Replica Selection Benchmark for C3 in a MEC environment with increased client concurrency.

We have also made a second benchmark allowing for more individual client concurrency just to guarantee that the results were not overly biased by the infrastructure as in the Infrastructure First approach. And as shown by Figure 5.8, we can see that they are not. This is because the algorithm *adjusted* itself to the circumstances. When using the Infrastructure First approach, it picked the infrastructure as the target replica if it was available, disregarding any type of metrics; this led to poor choices on the second benchmark where the outstanding requests to the infrastructure tracked by each client rose and the expected latency was a more pessimistic indicator. In the case of C3, it reacted to the increased outstanding requests and picked a different replica when the previous baseline would have decided otherwise.

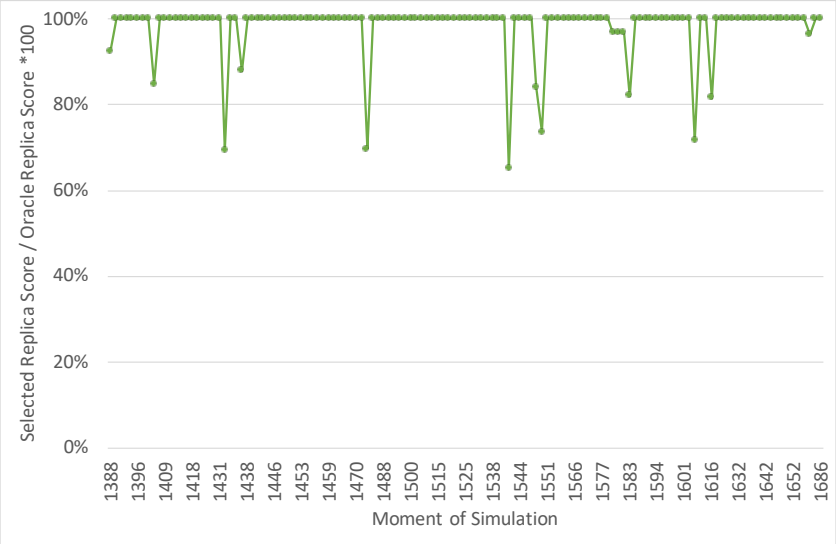


Figure 5.9: Ratio between the select replica score and the actual best replica score for the given download.

We can also see the gap closing on the score difference between the picked replica and the best amongst the available ones, as classifier by our criteria (Figure 5.9). The average replica selection holds around 95% of the score of the best option.

5.4.1.4 MEC C3

MEC C3 is still the C3 replica ranking scheme, but instead of disregarding unknown replicas and look at these as a last resort, it scores them with a neutral value (i.e. in the middle of the scale of possible scores) to try and place them between the best replicas and the least desirable ones. This allows for possibly better replicas to be picked much earlier and is thus a preferable behavior in MEC systems.

From Figure 5.10 we can see that this yields some positive results even though not drastically positive. The average replica selection quality rose from 93 to 95% and the quality of those selections system-wide kept in the previous 80%. Still, as we have seen in the previous section, there are additional improvement vectors to C3. We have once again observed around 40% replica availability on download and the score gaps were also similar.

We still observe some specially bad replica selections, such as the one at moment 1545. This is mainly caused by not taking power consumption into consideration. Also, being unaware of more than half of the existing replicas plays a big part.

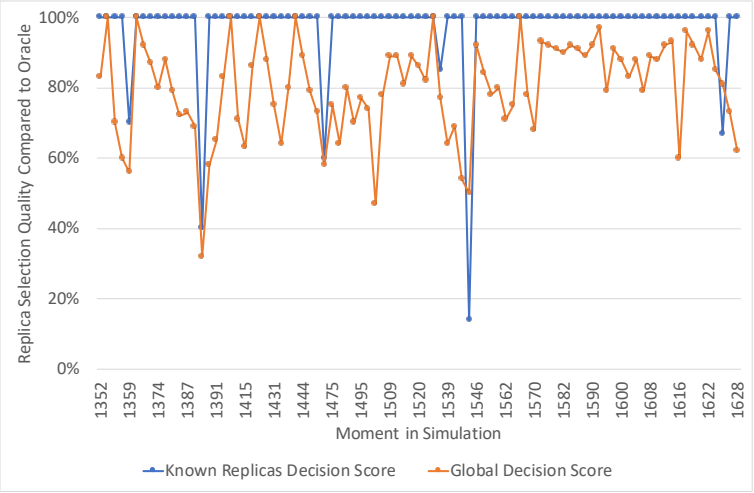


Figure 5.10: Ratio between the select replica score and the actual best replica score for the given download.

5.4.1.5 Wasabi

Finally, we present the results to our solution. Up until here we have seen the added benefit of using client- and server-side metrics to estimate the expected latency from each replica, which has resulted in more predictable selections. Still, there are several concerns that remain unattended, such as power consumption and metric values decadence. Wasabi gathers all these concerns into a single replica selection mechanism in order to improve MEC systems. The results are as presented in Figure 5.11.

First, looking at the replica selection from the available replicas, we see that it consistently picks the best replica. This was expected since the same specification is implemented on the evaluation side (the oracle) and we use the same information as the client had on its' state to rank the replicas. This was also a way for us to validate our algorithm: if the results were inconsistent, we would have to review both implementations. Essentially, this is what we have been comparing the other baselines to. The most interesting part of the chart is the replica selection quality considering the system snapshot at that moment. As we can see, not always did our algorithm pick the best replica according to the exact system state. And this is essentially because the client was not perceiving the whole system as it actually was. As in the other baselines, this is the common enemy. There are two important factors at play:

1. The client did not know the exact replicas of the target object at download time;

2. Some of the metrics stored at the client's selection module were already outdated or the client was missing some metrics.

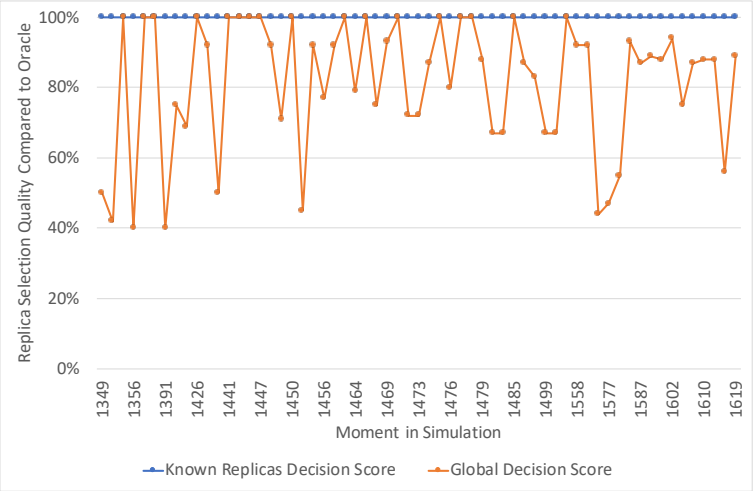


Figure 5.11: Replica Selection Benchmark for Wasabi

Regarding 1., we have once again estimated that, on average, the client was presented with about 40% of the existing replicas for each download. We have seen that it is a common trend among the simulations and which we will explain in the following section. Regarding 2., the quality of the selected replicas is directly proportional to the amount and freshness of the available information. In the best case scenario, the client would have an omniscient view of the system such as our post-processing tool which is able to reconstruct the exact system state at a certain moment. However, because we are talking about a distributed system, all we can do is try to improve the propagation of new metrics as well as the information about replicas being created or destroyed.

Nevertheless, we argue that the results are satisfactory as this benchmark had more than twice the number of best selections compared to any other, regarding the current system state. Also, most selections stayed around 80 to 100% while the remaining choices did not drop below 40%. On average, the replica selection quality was 82% in spite of considering only 40% of existing replicas, which is a big improvement compared to the previous solution in place, Infrastructure First.

5.4.1.6 Closing Remarks

In this section we have benchmarked all proposed baselines regarding their replica selection quality. We have presented the results in order of implementation complexity, starting from the Random Selection up until our final solution, Wasabi. We have seen improvements on each baseline over the previous one, the biggest leap being from Infrastructure First to C3 as it starts to use client- and server-side metrics to make better informed decisions.

In spite of our solution having by far the best consistency on picking the best replica,

it is still prone to pick a less favourable replica if the client does not have the most up-to-date information about the current system state. In our benchmarks this was observed mostly due to the available replicas (i.e. the replicas listed in the download notification) representing, on average, less than half of the already existing copies of the object. This phenomenon happened due to the way we wrote our traces, specially in the part of the past subscriptions. Here, nodes subscribe to the same tag all at once, with almost negligible intervals between each subscription and start downloading the same objects almost simultaneously and some are still downloading other objects from the future subscriptions. This is why most replicas do not appears listed in the available copies of the object download notification, because they are too recent and that information has not propagated yet. This can be a whole parallel topic as replica management is out of the scope of this thesis. In fact, the metadata update policy is something that was are part of Thyme GardenBed and is something we did not touch. Regardless, a system that would react that fast to this kinds of updates is probably not achievable. At most, the notification could also include the currently downloading nodes alongside the effectively confirmed replicas and the client could then try to be optimistic and contact one of those instead.

Still on up-to-date information, we have also noticed that already outdated metric readings we also behind some of the less satisfactory replica selections. This concerns our feedback mechanism and, just like the information about replica availability, there were times where our mechanism could not keep up with the pace of the trace execution. We are totally against adding adding new messages to the system specifically for this purpose and we are already making use of all existing messages to piggyback the necessary metrics. Thus, if we do not add extra vehicles of information then the solution might be adding extra information onto the current messages. We will be going through possible improvements in the next chapter. Nevertheless, we are satisfied by the current results.

5.4.2 System Reactivity

In the previous section we got a rough estimation of how reactive the whole system is. We have seen that it struggled to keep the list of available replicas up-to-date and some replica metrics were also outdated for some clients at download time. In this section we focus on the reactivity of the feedback system, i.e., how fast can a client update its metrics. To this end, we run the scenario described in Section 5.2 where we have a fixed number of mobile nodes plus one, replicate a set of objects through those same nodes and use the other node we left out to download those objects, resulting in multiple downloads with the same set of replicas. The time intervals between each download get smaller and smaller, and we abruptly drop the battery of one of the available nodes between downloads, to understand if the those changes can be perceived within that interval.

To recap on the important details of this test, all nodes start with 100% battery levels. There is a warm up phase to guarantee that they know each other, specifically to guarantee

that the node which will perform the evaluated downloads can know all its peers. After each download, we drop the battery levels of one of the replicas. The first two downloads have a 10 second interval between them, and this interval decreases 1 second on each consecutive download, being only 1 second between the last two, which was the time interval used between actions on the previous benchmarks.

What we were able to observe with this test was that, effectively, the downloading node got the metrics update up until the 2 seconds gap. For the last download, however, the available replicas were still sorted in the same way as in the previous one, which means that the update did not go through. Since we are using the system messages as a vehicle to broadcast these updates, we need to look into the messages exchanged between downloads. In this case, the only messages exchanged were the *Hello* messages, which are broadcast at a configured time rate (1 second in our configuration). This means that if no other messages are exchanged within the system, there is at least a chance for each client to update its replica metrics roughly each second. Those changes did not reflect on the last download because it happened to be processed first.

To confirm our hypothesis we reconfigured the *Hello* intervals to 5 seconds and indeed, this time around we only got about 50% success rate on the same test. It would have been interesting to exchange other kinds of actions between the downloads to see if these results could improve at the expense of other messages. However, this would make the test more complex and harder to reason about since these could introduce other delays and make the intervals between downloads be less accurate. Furthermore, it would be difficult for us to filter out the target downloads from the collected data.

Our conclusion is that the *Hello* messages have a great impact on metric updates. Shorter *Hello* intervals improve the overall freshness of system metrics. However, very short intervals on those messages can generate a lot of traffic, create bigger processing queues on the servers and ultimately punish battery-constrained devices. There should be a considerate trade-off when configuring the system.

5.4.3 System Overhead

In this section we evaluate the increase in byte count introduced by adding the metrics section the each system message. We run a decreased version of the first benchmark traces with only 16 nodes and register all the bytes sent by each node. After, we aggregate all these values to have a total amount of bytes transferred during the simulation. We run the simulation for our version of Thyme GardenBed and then repeat the process with a version that does not have our module integrated. In the end, we compare the two values.

From Figure 5.12 we can see that there is not an enormous overhead in communication when including the metrics payload in all system messages. Simulating the exact same scenario, we get an increase of 12% in the amount of bytes transferred. On average, each metric payload increased the size of each message by about 50 bytes. At this size, the metrics payload is smaller than any system message. However, it is still a considerable

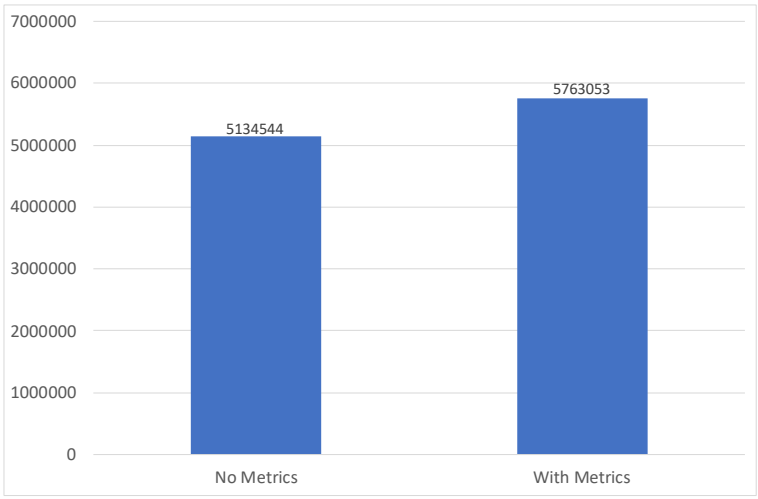


Figure 5.12: Bytes sent during simulation: no replica selection vs replica selection

size compared to the simplest messages, such as the *Hello* message. It is also important to note that in this trace the transferred files were always 1KB in size. If smaller files were to be transferred, the metrics payload would represent a slightly bigger percentage of the overal traffic.

As we have seen from Section 5.4.1, the less biased results for the Infrastructure First approach have shown that its replica selection quality is only 73% of Wasabi's when considering known replica metrics. Moreover, this approach has only scored 53% when looking at the true system state captured by the runtime snapshots, whereas Wasabi has scored 82%, which is an overall 29% better replica selection. This is a considerable increase which will reflect in the system's resource management and liveness. As we have seen, the cost of this benefit comes at a smaller rate, having the underlying feedback system increase the overall generated traffic by 12%. Still, the system can be fine-tuned to decrease this cost by filtering out the messages which less contribute for metrics dissemination in the *Metrics Aggregator*.

5.5 Final Remarks

In this chapter we have discussed what should be evaluated in our work. Then, we have defined an evaluation strategy which could yield the results we sought. After, we have discussed our experimental setup: the simulator, the execution traces and why simulation was required. Finally, we have presented the experimental results.

To understand how good the replica selection process in our solution is, we have defined some comparison baselines and classified their sorting of available replicas in two dimensions:

1. how much they tried to preserve and improve the MEC concerns laid out in Chapter 4, which are the foundation for our solution;

2. still with those concerns in mind, how those solutions would actually impact the real state of the system.

With this we have seen that there is a satisfactory improvement when dropping the previous solution and adopting Wasabi for selecting download replicas.

Additionally, we have seen that the additional payload injection for replica dissemination yields a considerable increase in the amount of transferred bytes. This, however, can be mitigated with a better tuning of the system by removing the metrics from the messages that less contribute to their dissemination and by decreasing the serialized representation of the the metrics payload. Ideally, the improvement yields should as big, or bigger than the percentage overhead.

Chapter 6

Conclusions

6.1 Conclusions

In this dissertation, we set out to build an adaptive replica selection mechanism that is able to improve MEC systems. We had to deal with the same concerns as the algorithms tailored for other environments, such as the cloud, but on top of that, we had to address other new challenges such as dealing with a dynamic set of replicas which is not known at start and is continuously changing over time. Furthermore, other concerns arose, such as energy efficiency due to power constrained devices, churn, less reliable communication and freshness of replica information.

To tackle this need, we have designed and developed Wasabi, an adaptive replica selection algorithm for MEC environments. To support the development and integration of Wasabi in existing MEC systems, we have developed a low-profile replica selection framework. This framework is system-agnostic and can not only be used in MEC systems, but actually in all kinds of distributed systems. We have also provided an integration of Wasabi with Thyme GardenBed, where the mobile nodes have symmetric responsibilities (both servers and clients) and the GardenBed infrastructure server was only integrated as a server.

From our experimental results we can proudly say that the integration of Wasabi within Thyme GardenBed, replacing the previous replica selection policy of always preferring the infrastructure server, has yield significant improvement on the quality of the selected replicas to retrieve data objects. We have also seen that the integration of the required feedback system between servers and clients creates a considerable overhead regarding the amount of data transferred between nodes, but which we believe can be easily minimized by applying some simple techniques.

Overall, we can say that we have accomplished the objectives for this dissertation.

6.2 System Improvements and Research Opportunities

Because there is always something else that can be added and improved, in this section we suggest some extensions and improvements to our work, and what kind of investigation can be conducted from here on.

6.2.1 Exploring Alternative Communication Protocols

This is a topic we have opened up in Section 2.3. Since currently the only communication medium supported by Thyme is WiFi, this topic arose in response to the question "*What if the congestion is in the AP*?". The idea is that mobile devices may consider alternative communication channels when they detect that it can improve the quality of service. In this sense, we have highlighted WiFi-Direct for direct peer-to-peer communication while offering a decent range of communication. Furthermore, it might be interesting to consider other protocols depending on device proximity, such as Bluetooth.

In Section 4.2.2.1 we have also suggested that the provided node identifier type when using our framework could not only be its unique identifier but rather the combination of the id with the communication protocol. With this, we could consider alternative communication channels to contact the same server, and even compare it with itself using different protocols.

The challenge here is in which communication channels should be considered and when. What to do when the device does not support that technology or protocol, how do we classify different communication channels and when do we have to establish those connections. Should we try to open all different channels as soon as possible and strive to maintain all connections or should these be established *ad hoc*?

6.2.2 Metric Dissemination Overhead

As seen from the results in Section 5.4.3, the dissemination of replica metrics adds a non-negligible overhead on the network layer, increasing the observed traffic size by 12%. Although there is a positive ratio between this increase and the replica selection quality increase, we believe we can further mitigate this side-effect of the underlying feedback system.

As it has already been suggested in Section 5.4.3, it could be interesting to understand how much each message type contributes for the dissemination of metrics and its impact in replica selection performance. We believe some messages could be filtered out of this process since they add little value to keep the overall system up-to-date but still contribute to the bandwidth penalties. We could even try to keep just the *Hello* messages as the vehicle for metrics dissemination, since we have seen from Section 5.4.2 that these alone were able to provide a high degree of freshness. Still, this freshness is tightly coupled with the periodicity of the *Hello* messages and the more frequent these are, the higher the overall byte count will be.

Bibliography

[1] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. "Mobile Edge Computing: A Survey." In: *IEEE Internet Things J.* 5.1 (2018), pp. 450–465. DOI: 10.1109/JIOT.2017.2750180. URL: https://doi.org/10.1109/JIOT.2017.2750180.

[2] M. T. Beck, M. Werner, S. Feld, and T. Schimper. "Mobile Edge Computing: A Taxonomy." In: *AFIN 2014 : The Sixth International Conference on Advances in Future Internet*. 2014, pp. 48–54. ISBN: 9781612083773.

[3] W. Jiang, H. Xie, X. Zhou, L. Fang, and J. Wang. "Performance Analysis and Improvement of Replica Selection Algorithms for Key-Value Stores." In: *2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, June 25-30, 2017*. Ed. by G. C. Fox. IEEE Computer Society, 2017, pp. 786–789. ISBN: 978-1-5386-1993-3. DOI: 10.1109/CLOUD.2017.115. URL: https://doi.org/10.1109/CLOUD.2017.115.

[4] Y. Lin, Y. Chen, G. Wang, and B. Deng. "Rigel: A Scalable and Lightweight Replica Selection Service for Replicated Distributed File System." In: *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid 2010, 17-20 May 2010, Melbourne, Victoria, Australia*. IEEE Computer Society, 2010, pp. 581–582. DOI: 10.1109/CCGRID.2010.51. URL: https://doi.org/10.1109/CCGRID.2010.51.

[5] M. Mamei and F. Zambonelli. "Programming Pervasive and Mobile Computing Applications with the TOTA Middleware." In: *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004), 14-17 March 2004, Orlando, FL, USA*. IEEE Computer Society, 2004, pp. 263–276. DOI: 10.1109/PERCOM.2004.1276864. URL: https://doi.org/10.1109/PERCOM.2004.1276864.

[6] G. Metri, A. Agrawal, R. Peri, and W. Shi. "What is eating up battery life on my SmartPhone: A case study." In: *International Conference on Energy Aware Computing, ICEAC 2012, Guzelyurt, Cyprus, December 3-5, 2012*. IEEE, 2012, pp. 1–6. DOI: 10.1109/ICEAC.2012.6471003. URL: https://doi.org/10.1109/ICEAC.2012.6471003.

[7] M. Mitzenmacher. "The Power of Two Choices in Randomized Load Balancing." In: *IEEE Trans. Parallel Distributed Syst.* 12.10 (2001), pp. 1094–1104. DOI: 10.1109/71.963420. URL: https://doi.org/10.1109/71.963420.

[8] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. "Making Sense of Performance in Data Analytics Frameworks." In: *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 2015, pp. 293–307. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout.

[9] 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 First ACM International Workshop on Wireless Sensor Networks and Applications, WSNA 2002, Atlanta, Georgia, USA, September 28, 2002*. Ed. by C. S. Raghavendra and K. M. Sivalingam. ACM, 2002, pp. 78–87. DOI: 10.1145/570738.570750. URL: https://doi.org/10.1145/570738.570750.

[10] *Riak Load Balancing and Proxy Configuration*. Last visited on July 2019. 2014. URL: http://docs.basho.com/riak/1.4.0/cookbooks/Load-Balancing-and-Proxy-Configuration/.

[11] J. A. Silva, F. Cerqueira, H. Paulino, J. M. Lourenço, J. Leitão, and N. M. Preguiça. "It's about Thyme: On the design and implementation of a time-aware reactive storage system for pervasive edge computing environments." In: *Future Gener. Comput. Syst.* 118 (2021), pp. 14–36. DOI: 10.1016/j.future.2020.12.008. URL: https://doi.org/10.1016/j.future.2020.12.008.

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

[13] J. A. Silva, H. Paulino, J. M. Lourenço, J. Leitão, and N. M. Preguiça. "Time-aware reactive storage in wireless edge environments." In: *MobiQuitous 2019, Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Houston, Texas, USA, November 12-14, 2019*. Ed. by H. V. Poor, Z. Han, D. Pompili, Z. Sun, and M. Pan. ACM, 2019, pp. 238–247. DOI: 10.1145/3360774.3360828. URL: https://doi.org/10.1145/3360774.3360828.

[14] 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 2020, Cork, Ireland, August 31 - September 3, 2020*. IEEE, 2020, pp. 40–49. ISBN: 978-1-7281-7374-0. DOI: 10.1109/WoWMoM49955.2020.00021. URL: https://doi.org/10.1109/WoWMoM49955.2020.00021.

[15] R. Simpson. *Mobile and tablet internet usage exceeds desktop for first time worldwide*. Last visited on July 2019. StatCounter. 2016. URL: http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide.

[16] Y. Su, D. Feng, Y. Hua, Z. Shi, and T. Zhu. "NetRS: Cutting Response Latency in Distributed Key-Value Stores with In-Network Replica Selection." In: *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. IEEE Computer Society, 2018, pp. 143–153. DOI: 10.1109/ICDCS.2018.00024. URL: https://doi.org/10.1109/ICDCS.2018.00024.

[17] P. L. Suresh, M. Canini, S. Schmid, and A. Feldmann. "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection." In: *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 2015, pp. 513–527. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/suresh.

[18] K. Thilakarathna, A. A. A. Karim, H. Petander, and A. Seneviratne. "MobiTribe: Enabling device centric social networking on smart mobile devices." In: *10th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, SECON 2013, New Orleans, LA, USA, 24-27 June, 2013*. IEEE, 2013, pp. 230–232. DOI: 10.1109/SAHCN.2013.6644982. URL: https://doi.org/10.1109/SAHCN.2013.6644982.

[19] K. Thilakarathna, H. Petander, J. Mestre, and A. Seneviratne. "MobiTribe: Cost Efficient Distributed User Generated Content Sharing on Smartphones." In: *IEEE Trans. Mob. Comput.* 13.9 (2014), pp. 2058–2070. DOI: 10.1109/TMC.2013.89. URL: https://doi.org/10.1109/TMC.2013.89.

[20] 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 2ndt ACM SIGCOMM Workshop on Networking, Systems, and Applications for Mobile Handhelds, MobiHeld 2010, New Delhi, India, August 30, 2010*. Ed. by L. P. Cox and A. Wolman. ACM, 2010, pp. 9–14. DOI: 10.1145/1851322.1851327. URL: https://doi.org/10.1145/1851322.1851327.

[21] P. Vieira. "A Persistent Publish/Subscribe System for Mobile Edge Computing." http://hdl.handle.net/10362/71124. Master's thesis. Faculty of Science and Technology, NOVA University of Lisbon, 2018.

[22] Y. Zhao and Y. Hu. "GRESS - a Grid Replica Selection Service." In: *Proceedings of the ISCA 16th International Conference on Parallel and Distributed Computing Systems, August 13-15, 2003, Atlantis Hotel, Reno, Nevada, USA*. Ed. by S. Yoo and H. Y. Youn. ISCA, 2003, pp. 423–429.

Annex I

Replica Selection Quality Trace

```
NODE$|$0$|$0
NODE$|$1$|$1
NODE$|$2$|$2
NODE$|$3$|$3
NODE$|$4$|$4
NODE$|$5$|$5
NODE$|$6$|$6
NODE$|$7$|$7
NODE$|$8$|$8
NODE$|$9$|$9
NODE$|$10$|$10
NODE$|$11$|$11
NODE$|$12$|$12
NODE$|$13$|$13
NODE$|$14$|$14
NODE$|$15$|$15
NODE$|$16$|$16
NODE$|$17$|$17
NODE$|$18$|$18
NODE$|$19$|$19
NODE$|$20$|$20
NODE$|$21$|$21
NODE$|$22$|$22
NODE$|$23$|$23
NODE$|$24$|$24
NODE$|$25$|$25
NODE$|$26$|$26
NODE$|$27$|$27
NODE$|$28$|$28
NODE$|$29$|$29
NODE$|$30$|$30
NODE$|$31$|$31
NODE$|$32$|$32
NODE$|$33$|$33
NODE$|$34$|$34
```

```
NODE$|$35$|$35
NODE$|$36$|$36
NODE$|$37$|$37
NODE$|$38$|$38
NODE$|$39$|$39
NODE$|$40$|$40
NODE$|$41$|$41
NODE$|$42$|$42
NODE$|$43$|$43
NODE$|$44$|$44
NODE$|$45$|$45
NODE$|$46$|$46
NODE$|$47$|$47
NODE$|$48$|$48
NODE$|$49$|$49
NODE$|$50$|$50
NODE$|$51$|$51
NODE$|$52$|$52
NODE$|$53$|$53
NODE$|$54$|$54
NODE$|$55$|$55
NODE$|$56$|$56
NODE$|$57$|$57
NODE$|$58$|$58
NODE$|$59$|$59
NODE$|$60$|$60
NODE$|$61$|$61
NODE$|$62$|$62
NODE$|$63$|$63
SUB_F$|$184$|$1$|$TAG0
SUB_F$|$185$|$6$|$TAG0
SUB_F$|$186$|$7$|$TAG0
SUB_F$|$187$|$12$|$TAG0
SUB_F$|$188$|$14$|$TAG0
SUB_F$|$189$|$19$|$TAG0
SUB_F$|$190$|$21$|$TAG0
SUB_F$|$191$|$22$|$TAG0
SUB_F$|$192$|$23$|$TAG0
SUB_F$|$193$|$25$|$TAG0
SUB_F$|$194$|$26$|$TAG0
SUB_F$|$195$|$29$|$TAG0
SUB_F$|$196$|$30$|$TAG0
SUB_F$|$197$|$32$|$TAG0
SUB_F$|$198$|$33$|$TAG0
SUB_F$|$199$|$34$|$TAG0
SUB_F$|$200$|$35$|$TAG0
SUB_F$|$201$|$37$|$TAG0
SUB_F$|$202$|$38$|$TAG0
SUB_F$|$203$|$39$|$TAG0
SUB_F$|$204$|$41$|$TAG0
```

```
SUB_F$|$205$|$42$|$TAG0
SUB_F$|$206$|$44$|$TAG0
SUB_F$|$207$|$45$|$TAG0
SUB_F$|$208$|$46$|$TAG0
SUB_F$|$209$|$51$|$TAG0
SUB_F$|$210$|$54$|$TAG0
SUB_F$|$211$|$55$|$TAG0
SUB_F$|$212$|$56$|$TAG0
SUB_F$|$213$|$60$|$TAG0
SUB_F$|$214$|$0$|$TAG1
SUB_F$|$215$|$1$|$TAG1
SUB_F$|$216$|$2$|$TAG1
SUB_F$|$217$|$3$|$TAG1
SUB_F$|$218$|$4$|$TAG1
SUB_F$|$219$|$5$|$TAG1
SUB_F$|$220$|$6$|$TAG1
SUB_F$|$221$|$12$|$TAG1
SUB_F$|$222$|$13$|$TAG1
SUB_F$|$223$|$15$|$TAG1
SUB_F$|$224$|$17$|$TAG1
SUB_F$|$225$|$19$|$TAG1
SUB_F$|$226$|$21$|$TAG1
SUB_F$|$227$|$22$|$TAG1
SUB_F$|$228$|$24$|$TAG1
SUB_F$|$229$|$27$|$TAG1
SUB_F$|$230$|$28$|$TAG1
SUB_F$|$231$|$30$|$TAG1
SUB_F$|$232$|$31$|$TAG1
SUB_F$|$233$|$33$|$TAG1
SUB_F$|$234$|$36$|$TAG1
SUB_F$|$235$|$37$|$TAG1
SUB_F$|$236$|$38$|$TAG1
SUB_F$|$237$|$42$|$TAG1
SUB_F$|$238$|$43$|$TAG1
SUB_F$|$239$|$44$|$TAG1
SUB_F$|$240$|$45$|$TAG1
SUB_F$|$241$|$46$|$TAG1
SUB_F$|$242$|$47$|$TAG1
SUB_F$|$243$|$50$|$TAG1
SUB_F$|$244$|$51$|$TAG1
SUB_F$|$245$|$52$|$TAG1
SUB_F$|$246$|$54$|$TAG1
SUB_F$|$247$|$55$|$TAG1
SUB_F$|$248$|$57$|$TAG1
SUB_F$|$249$|$58$|$TAG1
SUB_F$|$250$|$59$|$TAG1
SUB_F$|$251$|$61$|$TAG1
SUB_F$|$252$|$62$|$TAG1
SUB_F$|$253$|$63$|$TAG1
SUB_F$|$254$|$0$|$TAG2
```

```
SUB_F$|$255$|$1$|$TAG2
SUB_F$|$256$|$2$|$TAG2
SUB_F$|$257$|$4$|$TAG2
SUB_F$|$258$|$5$|$TAG2
SUB_F$|$259$|$8$|$TAG2
SUB_F$|$260$|$12$|$TAG2
SUB_F$|$261$|$15$|$TAG2
SUB_F$|$262$|$16$|$TAG2
SUB_F$|$263$|$18$|$TAG2
SUB_F$|$264$|$19$|$TAG2
SUB_F$|$265$|$20$|$TAG2
SUB_F$|$266$|$21$|$TAG2
SUB_F$|$267$|$24$|$TAG2
SUB_F$|$268$|$25$|$TAG2
SUB_F$|$269$|$26$|$TAG2
SUB_F$|$270$|$27$|$TAG2
SUB_F$|$271$|$28$|$TAG2
SUB_F$|$272$|$29$|$TAG2
SUB_F$|$273$|$30$|$TAG2
SUB_F$|$274$|$33$|$TAG2
SUB_F$|$275$|$34$|$TAG2
SUB_F$|$276$|$38$|$TAG2
SUB_F$|$277$|$39$|$TAG2
SUB_F$|$278$|$46$|$TAG2
SUB_F$|$279$|$49$|$TAG2
SUB_F$|$280$|$52$|$TAG2
SUB_F$|$281$|$53$|$TAG2
SUB_F$|$282$|$54$|$TAG2
SUB_F$|$283$|$55$|$TAG2
SUB_F$|$284$|$56$|$TAG2
SUB_F$|$285$|$1$|$TAG3
SUB_F$|$286$|$3$|$TAG3
SUB_F$|$287$|$5$|$TAG3
SUB_F$|$288$|$8$|$TAG3
SUB_F$|$289$|$10$|$TAG3
SUB_F$|$290$|$11$|$TAG3
SUB_F$|$291$|$14$|$TAG3
SUB_F$|$292$|$15$|$TAG3
SUB_F$|$293$|$16$|$TAG3
SUB_F$|$294$|$17$|$TAG3
SUB_F$|$295$|$19$|$TAG3
SUB_F$|$296$|$27$|$TAG3
SUB_F$|$297$|$31$|$TAG3
SUB_F$|$298$|$34$|$TAG3
SUB_F$|$299$|$36$|$TAG3
SUB_F$|$300$|$38$|$TAG3
SUB_F$|$301$|$39$|$TAG3
SUB_F$|$302$|$40$|$TAG3
SUB_F$|$303$|$41$|$TAG3
SUB_F$|$304$|$43$|$TAG3
```

```
SUB_F$|$305$|$45$|$TAG3
SUB_F$|$306$|$46$|$TAG3
SUB_F$|$307$|$47$|$TAG3
SUB_F$|$308$|$49$|$TAG3
SUB_F$|$309$|$51$|$TAG3
SUB_F$|$310$|$52$|$TAG3
SUB_F$|$311$|$53$|$TAG3
SUB_F$|$312$|$56$|$TAG3
SUB_F$|$313$|$59$|$TAG3
SUB_F$|$314$|$0$|$TAG4
SUB_F$|$315$|$2$|$TAG4
SUB_F$|$316$|$4$|$TAG4
SUB_F$|$317$|$7$|$TAG4
SUB_F$|$318$|$12$|$TAG4
SUB_F$|$319$|$16$|$TAG4
SUB_F$|$320$|$24$|$TAG4
SUB_F$|$321$|$25$|$TAG4
SUB_F$|$322$|$26$|$TAG4
SUB_F$|$323$|$28$|$TAG4
SUB_F$|$324$|$30$|$TAG4
SUB_F$|$325$|$32$|$TAG4
SUB_F$|$326$|$33$|$TAG4
SUB_F$|$327$|$34$|$TAG4
SUB_F$|$328$|$36$|$TAG4
SUB_F$|$329$|$37$|$TAG4
SUB_F$|$330$|$38$|$TAG4
SUB_F$|$331$|$39$|$TAG4
SUB_F$|$332$|$42$|$TAG4
SUB_F$|$333$|$43$|$TAG4
SUB_F$|$334$|$44$|$TAG4
SUB_F$|$335$|$45$|$TAG4
SUB_F$|$336$|$46$|$TAG4
SUB_F$|$337$|$47$|$TAG4
SUB_F$|$338$|$50$|$TAG4
SUB_F$|$339$|$54$|$TAG4
SUB_F$|$340$|$55$|$TAG4
SUB_F$|$341$|$57$|$TAG4
SUB_F$|$342$|$58$|$TAG4
SUB_F$|$343$|$60$|$TAG4
SUB_F$|$344$|$62$|$TAG4
SUB_F$|$345$|$1$|$TAG5
SUB_F$|$346$|$4$|$TAG5
SUB_F$|$347$|$5$|$TAG5
SUB_F$|$348$|$7$|$TAG5
SUB_F$|$349$|$11$|$TAG5
SUB_F$|$350$|$13$|$TAG5
SUB_F$|$351$|$15$|$TAG5
SUB_F$|$352$|$17$|$TAG5
SUB_F$|$353$|$19$|$TAG5
SUB_F$|$354$|$24$|$TAG5
```

```
SUB_F$|$355$|$26$|$TAG5
SUB_F$|$356$|$27$|$TAG5
SUB_F$|$357$|$29$|$TAG5
SUB_F$|$358$|$31$|$TAG5
SUB_F$|$359$|$33$|$TAG5
SUB_F$|$360$|$36$|$TAG5
SUB_F$|$361$|$37$|$TAG5
SUB_F$|$362$|$42$|$TAG5
SUB_F$|$363$|$43$|$TAG5
SUB_F$|$364$|$45$|$TAG5
SUB_F$|$365$|$48$|$TAG5
SUB_F$|$366$|$49$|$TAG5
SUB_F$|$367$|$50$|$TAG5
SUB_F$|$368$|$51$|$TAG5
SUB_F$|$369$|$52$|$TAG5
SUB_F$|$370$|$53$|$TAG5
SUB_F$|$371$|$55$|$TAG5
SUB_F$|$372$|$56$|$TAG5
SUB_F$|$373$|$57$|$TAG5
SUB_F$|$374$|$58$|$TAG5
SUB_F$|$375$|$59$|$TAG5
SUB_F$|$376$|$60$|$TAG5
SUB_F$|$377$|$61$|$TAG5
SUB_F$|$378$|$62$|$TAG5
SUB_F$|$379$|$63$|$TAG5
SUB_F$|$380$|$0$|$TAG6
SUB_F$|$381$|$3$|$TAG6
SUB_F$|$382$|$8$|$TAG6
SUB_F$|$383$|$16$|$TAG6
SUB_F$|$384$|$19$|$TAG6
SUB_F$|$385$|$20$|$TAG6
SUB_F$|$386$|$23$|$TAG6
SUB_F$|$387$|$24$|$TAG6
SUB_F$|$388$|$25$|$TAG6
SUB_F$|$389$|$29$|$TAG6
SUB_F$|$390$|$30$|$TAG6
SUB_F$|$391$|$31$|$TAG6
SUB_F$|$392$|$35$|$TAG6
SUB_F$|$393$|$36$|$TAG6
SUB_F$|$394$|$41$|$TAG6
SUB_F$|$395$|$42$|$TAG6
SUB_F$|$396$|$43$|$TAG6
SUB_F$|$397$|$45$|$TAG6
SUB_F$|$398$|$49$|$TAG6
SUB_F$|$399$|$50$|$TAG6
SUB_F$|$400$|$53$|$TAG6
SUB_F$|$401$|$54$|$TAG6
SUB_F$|$402$|$55$|$TAG6
SUB_F$|$403$|$57$|$TAG6
SUB_F$|$404$|$58$|$TAG6
```

```
SUB_F$|$405$|$62$|$TAG6
SUB_F$|$406$|$63$|$TAG6
SUB_F$|$407$|$3$|$TAG7
SUB_F$|$408$|$7$|$TAG7
SUB_F$|$409$|$9$|$TAG7
SUB_F$|$410$|$11$|$TAG7
SUB_F$|$411$|$14$|$TAG7
SUB_F$|$412$|$15$|$TAG7
SUB_F$|$413$|$19$|$TAG7
SUB_F$|$414$|$20$|$TAG7
SUB_F$|$415$|$21$|$TAG7
SUB_F$|$416$|$22$|$TAG7
SUB_F$|$417$|$27$|$TAG7
SUB_F$|$418$|$30$|$TAG7
SUB_F$|$419$|$31$|$TAG7
SUB_F$|$420$|$32$|$TAG7
SUB_F$|$421$|$35$|$TAG7
SUB_F$|$422$|$41$|$TAG7
SUB_F$|$423$|$42$|$TAG7
SUB_F$|$424$|$43$|$TAG7
SUB_F$|$425$|$45$|$TAG7
SUB_F$|$426$|$46$|$TAG7
SUB_F$|$427$|$47$|$TAG7
SUB_F$|$428$|$50$|$TAG7
SUB_F$|$429$|$53$|$TAG7
SUB_F$|$430$|$54$|$TAG7
SUB_F$|$431$|$55$|$TAG7
SUB_F$|$432$|$56$|$TAG7
PUB$|$463$|$3$|$466$|$Object 466$|$Published by 3$|$TAG0
PUB$|$464$|$7$|$471$|$Object 471$|$Published by 7$|$TAG0
PUB$|$465$|$8$|$473$|$Object 473$|$Published by 8$|$TAG0
PUB$|$466$|$9$|$475$|$Object 475$|$Published by 9$|$TAG0
PUB$|$467$|$11$|$478$|$Object 478$|$Published by 11$|$TAG0
PUB$|$468$|$12$|$480$|$Object 480$|$Published by 12$|$TAG0
PUB$|$469$|$13$|$482$|$Object 482$|$Published by 13$|$TAG0
PUB$|$470$|$15$|$485$|$Object 485$|$Published by 15$|$TAG0
PUB$|$471$|$17$|$488$|$Object 488$|$Published by 17$|$TAG0
PUB$|$472$|$19$|$491$|$Object 491$|$Published by 19$|$TAG0
PUB$|$473$|$21$|$494$|$Object 494$|$Published by 21$|$TAG0
PUB$|$474$|$22$|$496$|$Object 496$|$Published by 22$|$TAG0
PUB$|$475$|$24$|$499$|$Object 499$|$Published by 24$|$TAG0
PUB$|$476$|$29$|$505$|$Object 505$|$Published by 29$|$TAG0
PUB$|$477$|$36$|$513$|$Object 513$|$Published by 36$|$TAG0
PUB$|$478$|$37$|$515$|$Object 515$|$Published by 37$|$TAG0
PUB$|$479$|$40$|$519$|$Object 519$|$Published by 40$|$TAG0
PUB$|$480$|$41$|$521$|$Object 521$|$Published by 41$|$TAG0
PUB$|$481$|$43$|$524$|$Object 524$|$Published by 43$|$TAG0
PUB$|$482$|$45$|$527$|$Object 527$|$Published by 45$|$TAG0
PUB$|$483$|$47$|$530$|$Object 530$|$Published by 47$|$TAG0
PUB$|$484$|$48$|$532$|$Object 532$|$Published by 48$|$TAG0
```

```
PUB$|$485$|$49$|$534$|$Object 534$|$Published by 49$|$TAG0
PUB$|$486$|$52$|$538$|$Object 538$|$Published by 52$|$TAG0
PUB$|$487$|$53$|$540$|$Object 540$|$Published by 53$|$TAG0
PUB$|$488$|$54$|$542$|$Object 542$|$Published by 54$|$TAG0
PUB$|$489$|$63$|$552$|$Object 552$|$Published by 63$|$TAG0
PUB$|$490$|$0$|$490$|$Object 490$|$Published by 0$|$TAG1
PUB$|$491$|$1$|$492$|$Object 492$|$Published by 1$|$TAG1
PUB$|$492$|$3$|$495$|$Object 495$|$Published by 3$|$TAG1
PUB$|$493$|$4$|$497$|$Object 497$|$Published by 4$|$TAG1
PUB$|$494$|$5$|$499$|$Object 499$|$Published by 5$|$TAG1
PUB$|$495$|$7$|$502$|$Object 502$|$Published by 7$|$TAG1
PUB$|$496$|$10$|$506$|$Object 506$|$Published by 10$|$TAG1
PUB$|$497$|$11$|$508$|$Object 508$|$Published by 11$|$TAG1
PUB$|$498$|$12$|$510$|$Object 510$|$Published by 12$|$TAG1
PUB$|$499$|$14$|$513$|$Object 513$|$Published by 14$|$TAG1
PUB$|$500$|$18$|$518$|$Object 518$|$Published by 18$|$TAG1
PUB$|$501$|$20$|$521$|$Object 521$|$Published by 20$|$TAG1
PUB$|$502$|$22$|$524$|$Object 524$|$Published by 22$|$TAG1
PUB$|$503$|$23$|$526$|$Object 526$|$Published by 23$|$TAG1
PUB$|$504$|$25$|$529$|$Object 529$|$Published by 25$|$TAG1
PUB$|$505$|$28$|$533$|$Object 533$|$Published by 28$|$TAG1
PUB$|$506$|$31$|$537$|$Object 537$|$Published by 31$|$TAG1
PUB$|$507$|$34$|$541$|$Object 541$|$Published by 34$|$TAG1
PUB$|$508$|$36$|$544$|$Object 544$|$Published by 36$|$TAG1
PUB$|$509$|$38$|$547$|$Object 547$|$Published by 38$|$TAG1
PUB$|$510$|$39$|$549$|$Object 549$|$Published by 39$|$TAG1
PUB$|$511$|$40$|$551$|$Object 551$|$Published by 40$|$TAG1
PUB$|$512$|$41$|$553$|$Object 553$|$Published by 41$|$TAG1
PUB$|$513$|$42$|$555$|$Object 555$|$Published by 42$|$TAG1
PUB$|$514$|$43$|$557$|$Object 557$|$Published by 43$|$TAG1
PUB$|$515$|$44$|$559$|$Object 559$|$Published by 44$|$TAG1
PUB$|$516$|$49$|$565$|$Object 565$|$Published by 49$|$TAG1
PUB$|$517$|$50$|$567$|$Object 567$|$Published by 50$|$TAG1
PUB$|$518$|$52$|$570$|$Object 570$|$Published by 52$|$TAG1
PUB$|$519$|$53$|$572$|$Object 572$|$Published by 53$|$TAG1
PUB$|$520$|$60$|$580$|$Object 580$|$Published by 60$|$TAG1
PUB$|$521$|$61$|$582$|$Object 582$|$Published by 61$|$TAG1
PUB$|$522$|$62$|$584$|$Object 584$|$Published by 62$|$TAG1
PUB$|$523$|$63$|$586$|$Object 586$|$Published by 63$|$TAG1
PUB$|$524$|$2$|$526$|$Object 526$|$Published by 2$|$TAG2
PUB$|$525$|$3$|$528$|$Object 528$|$Published by 3$|$TAG2
PUB$|$526$|$4$|$530$|$Object 530$|$Published by 4$|$TAG2
PUB$|$527$|$7$|$534$|$Object 534$|$Published by 7$|$TAG2
PUB$|$528$|$8$|$536$|$Object 536$|$Published by 8$|$TAG2
PUB$|$529$|$12$|$541$|$Object 541$|$Published by 12$|$TAG2
PUB$|$530$|$14$|$544$|$Object 544$|$Published by 14$|$TAG2
PUB$|$531$|$15$|$546$|$Object 546$|$Published by 15$|$TAG2
PUB$|$532$|$20$|$552$|$Object 552$|$Published by 20$|$TAG2
PUB$|$533$|$21$|$554$|$Object 554$|$Published by 21$|$TAG2
PUB$|$534$|$22$|$556$|$Object 556$|$Published by 22$|$TAG2
```

```
PUB$|$535$|$24$|$559$|$Object 559$|$Published by 24$|$TAG2
PUB$|$536$|$28$|$564$|$Object 564$|$Published by 28$|$TAG2
PUB$|$537$|$29$|$566$|$Object 566$|$Published by 29$|$TAG2
PUB$|$538$|$33$|$571$|$Object 571$|$Published by 33$|$TAG2
PUB$|$539$|$37$|$576$|$Object 576$|$Published by 37$|$TAG2
PUB$|$540$|$39$|$579$|$Object 579$|$Published by 39$|$TAG2
PUB$|$541$|$43$|$584$|$Object 584$|$Published by 43$|$TAG2
PUB$|$542$|$46$|$588$|$Object 588$|$Published by 46$|$TAG2
PUB$|$543$|$48$|$591$|$Object 591$|$Published by 48$|$TAG2
PUB$|$544$|$51$|$595$|$Object 595$|$Published by 51$|$TAG2
PUB$|$545$|$53$|$598$|$Object 598$|$Published by 53$|$TAG2
PUB$|$546$|$54$|$600$|$Object 600$|$Published by 54$|$TAG2
PUB$|$547$|$55$|$602$|$Object 602$|$Published by 55$|$TAG2
PUB$|$548$|$60$|$608$|$Object 608$|$Published by 60$|$TAG2
PUB$|$549$|$62$|$611$|$Object 611$|$Published by 62$|$TAG2
PUB$|$550$|$63$|$613$|$Object 613$|$Published by 63$|$TAG2
PUB$|$551$|$0$|$551$|$Object 551$|$Published by 0$|$TAG3
PUB$|$552$|$1$|$553$|$Object 553$|$Published by 1$|$TAG3
PUB$|$553$|$7$|$560$|$Object 560$|$Published by 7$|$TAG3
PUB$|$554$|$9$|$563$|$Object 563$|$Published by 9$|$TAG3
PUB$|$555$|$10$|$565$|$Object 565$|$Published by 10$|$TAG3
PUB$|$556$|$15$|$571$|$Object 571$|$Published by 15$|$TAG3
PUB$|$557$|$16$|$573$|$Object 573$|$Published by 16$|$TAG3
PUB$|$558$|$18$|$576$|$Object 576$|$Published by 18$|$TAG3
PUB$|$559$|$20$|$579$|$Object 579$|$Published by 20$|$TAG3
PUB$|$560$|$21$|$581$|$Object 581$|$Published by 21$|$TAG3
PUB$|$561$|$22$|$583$|$Object 583$|$Published by 22$|$TAG3
PUB$|$562$|$23$|$585$|$Object 585$|$Published by 23$|$TAG3
PUB$|$563$|$24$|$587$|$Object 587$|$Published by 24$|$TAG3
PUB$|$564$|$30$|$594$|$Object 594$|$Published by 30$|$TAG3
PUB$|$565$|$31$|$596$|$Object 596$|$Published by 31$|$TAG3
PUB$|$566$|$34$|$600$|$Object 600$|$Published by 34$|$TAG3
PUB$|$567$|$37$|$604$|$Object 604$|$Published by 37$|$TAG3
PUB$|$568$|$43$|$611$|$Object 611$|$Published by 43$|$TAG3
PUB$|$569$|$45$|$614$|$Object 614$|$Published by 45$|$TAG3
PUB$|$570$|$46$|$616$|$Object 616$|$Published by 46$|$TAG3
PUB$|$571$|$50$|$621$|$Object 621$|$Published by 50$|$TAG3
PUB$|$572$|$52$|$624$|$Object 624$|$Published by 52$|$TAG3
PUB$|$573$|$54$|$627$|$Object 627$|$Published by 54$|$TAG3
PUB$|$574$|$55$|$629$|$Object 629$|$Published by 55$|$TAG3
PUB$|$575$|$56$|$631$|$Object 631$|$Published by 56$|$TAG3
PUB$|$576$|$57$|$633$|$Object 633$|$Published by 57$|$TAG3
PUB$|$577$|$58$|$635$|$Object 635$|$Published by 58$|$TAG3
PUB$|$578$|$59$|$637$|$Object 637$|$Published by 59$|$TAG3
PUB$|$579$|$1$|$580$|$Object 580$|$Published by 1$|$TAG4
PUB$|$580$|$3$|$583$|$Object 583$|$Published by 3$|$TAG4
PUB$|$581$|$8$|$589$|$Object 589$|$Published by 8$|$TAG4
PUB$|$582$|$11$|$593$|$Object 593$|$Published by 11$|$TAG4
PUB$|$583$|$12$|$595$|$Object 595$|$Published by 12$|$TAG4
PUB$|$584$|$14$|$598$|$Object 598$|$Published by 14$|$TAG4
```

```
PUB$|$585$|$15$|$600$|$Object 600$|$Published by 15$|$TAG4
PUB$|$586$|$16$|$602$|$Object 602$|$Published by 16$|$TAG4
PUB$|$587$|$17$|$604$|$Object 604$|$Published by 17$|$TAG4
PUB$|$588$|$20$|$608$|$Object 608$|$Published by 20$|$TAG4
PUB$|$589$|$21$|$610$|$Object 610$|$Published by 21$|$TAG4
PUB$|$590$|$23$|$613$|$Object 613$|$Published by 23$|$TAG4
PUB$|$591$|$24$|$615$|$Object 615$|$Published by 24$|$TAG4
PUB$|$592$|$26$|$618$|$Object 618$|$Published by 26$|$TAG4
PUB$|$593$|$28$|$621$|$Object 621$|$Published by 28$|$TAG4
PUB$|$594$|$30$|$624$|$Object 624$|$Published by 30$|$TAG4
PUB$|$595$|$31$|$626$|$Object 626$|$Published by 31$|$TAG4
PUB$|$596$|$34$|$630$|$Object 630$|$Published by 34$|$TAG4
PUB$|$597$|$40$|$637$|$Object 637$|$Published by 40$|$TAG4
PUB$|$598$|$46$|$644$|$Object 644$|$Published by 46$|$TAG4
PUB$|$599$|$48$|$647$|$Object 647$|$Published by 48$|$TAG4
PUB$|$600$|$49$|$649$|$Object 649$|$Published by 49$|$TAG4
PUB$|$601$|$53$|$654$|$Object 654$|$Published by 53$|$TAG4
PUB$|$602$|$54$|$656$|$Object 656$|$Published by 54$|$TAG4
PUB$|$603$|$56$|$659$|$Object 659$|$Published by 56$|$TAG4
PUB$|$604$|$58$|$662$|$Object 662$|$Published by 58$|$TAG4
PUB$|$605$|$59$|$664$|$Object 664$|$Published by 59$|$TAG4
PUB$|$606$|$60$|$666$|$Object 666$|$Published by 60$|$TAG4
PUB$|$607$|$61$|$668$|$Object 668$|$Published by 61$|$TAG4
PUB$|$608$|$62$|$670$|$Object 670$|$Published by 62$|$TAG4
PUB$|$609$|$3$|$612$|$Object 612$|$Published by 3$|$TAG5
PUB$|$610$|$7$|$617$|$Object 617$|$Published by 7$|$TAG5
PUB$|$611$|$9$|$620$|$Object 620$|$Published by 9$|$TAG5
PUB$|$612$|$11$|$623$|$Object 623$|$Published by 11$|$TAG5
PUB$|$613$|$12$|$625$|$Object 625$|$Published by 12$|$TAG5
PUB$|$614$|$13$|$627$|$Object 627$|$Published by 13$|$TAG5
PUB$|$615$|$14$|$629$|$Object 629$|$Published by 14$|$TAG5
PUB$|$616$|$17$|$633$|$Object 633$|$Published by 17$|$TAG5
PUB$|$617$|$25$|$642$|$Object 642$|$Published by 25$|$TAG5
PUB$|$618$|$27$|$645$|$Object 645$|$Published by 27$|$TAG5
PUB$|$619$|$29$|$648$|$Object 648$|$Published by 29$|$TAG5
PUB$|$620$|$34$|$654$|$Object 654$|$Published by 34$|$TAG5
PUB$|$621$|$37$|$658$|$Object 658$|$Published by 37$|$TAG5
PUB$|$622$|$38$|$660$|$Object 660$|$Published by 38$|$TAG5
PUB$|$623$|$41$|$664$|$Object 664$|$Published by 41$|$TAG5
PUB$|$624$|$42$|$666$|$Object 666$|$Published by 42$|$TAG5
PUB$|$625$|$45$|$670$|$Object 670$|$Published by 45$|$TAG5
PUB$|$626$|$52$|$678$|$Object 678$|$Published by 52$|$TAG5
PUB$|$627$|$53$|$680$|$Object 680$|$Published by 53$|$TAG5
PUB$|$628$|$54$|$682$|$Object 682$|$Published by 54$|$TAG5
PUB$|$629$|$55$|$684$|$Object 684$|$Published by 55$|$TAG5
PUB$|$630$|$56$|$686$|$Object 686$|$Published by 56$|$TAG5
PUB$|$631$|$58$|$689$|$Object 689$|$Published by 58$|$TAG5
PUB$|$632$|$59$|$691$|$Object 691$|$Published by 59$|$TAG5
PUB$|$633$|$61$|$694$|$Object 694$|$Published by 61$|$TAG5
PUB$|$634$|$63$|$697$|$Object 697$|$Published by 63$|$TAG5
```

```
PUB$|$635$|$0$|$635$|$Object 635$|$Published by 0$|$TAG6
PUB$|$636$|$1$|$637$|$Object 637$|$Published by 1$|$TAG6
PUB$|$637$|$3$|$640$|$Object 640$|$Published by 3$|$TAG6
PUB$|$638$|$5$|$643$|$Object 643$|$Published by 5$|$TAG6
PUB$|$639$|$14$|$653$|$Object 653$|$Published by 14$|$TAG6
PUB$|$640$|$15$|$655$|$Object 655$|$Published by 15$|$TAG6
PUB$|$641$|$16$|$657$|$Object 657$|$Published by 16$|$TAG6
PUB$|$642$|$17$|$659$|$Object 659$|$Published by 17$|$TAG6
PUB$|$643$|$18$|$661$|$Object 661$|$Published by 18$|$TAG6
PUB$|$644$|$19$|$663$|$Object 663$|$Published by 19$|$TAG6
PUB$|$645$|$23$|$668$|$Object 668$|$Published by 23$|$TAG6
PUB$|$646$|$24$|$670$|$Object 670$|$Published by 24$|$TAG6
PUB$|$647$|$25$|$672$|$Object 672$|$Published by 25$|$TAG6
PUB$|$648$|$26$|$674$|$Object 674$|$Published by 26$|$TAG6
PUB$|$649$|$28$|$677$|$Object 677$|$Published by 28$|$TAG6
PUB$|$650$|$35$|$685$|$Object 685$|$Published by 35$|$TAG6
PUB$|$651$|$37$|$688$|$Object 688$|$Published by 37$|$TAG6
PUB$|$652$|$39$|$691$|$Object 691$|$Published by 39$|$TAG6
PUB$|$653$|$41$|$694$|$Object 694$|$Published by 41$|$TAG6
PUB$|$654$|$42$|$696$|$Object 696$|$Published by 42$|$TAG6
PUB$|$655$|$44$|$699$|$Object 699$|$Published by 44$|$TAG6
PUB$|$656$|$47$|$703$|$Object 703$|$Published by 47$|$TAG6
PUB$|$657$|$50$|$707$|$Object 707$|$Published by 50$|$TAG6
PUB$|$658$|$54$|$712$|$Object 712$|$Published by 54$|$TAG6
PUB$|$659$|$58$|$717$|$Object 717$|$Published by 58$|$TAG6
PUB$|$660$|$59$|$719$|$Object 719$|$Published by 59$|$TAG6
PUB$|$661$|$60$|$721$|$Object 721$|$Published by 60$|$TAG6
PUB$|$662$|$61$|$723$|$Object 723$|$Published by 61$|$TAG6
PUB$|$663$|$62$|$725$|$Object 725$|$Published by 62$|$TAG6
PUB$|$664$|$63$|$727$|$Object 727$|$Published by 63$|$TAG6
PUB$|$665$|$0$|$665$|$Object 665$|$Published by 0$|$TAG7
PUB$|$666$|$1$|$667$|$Object 667$|$Published by 1$|$TAG7
PUB$|$667$|$9$|$676$|$Object 676$|$Published by 9$|$TAG7
PUB$|$668$|$11$|$679$|$Object 679$|$Published by 11$|$TAG7
PUB$|$669$|$12$|$681$|$Object 681$|$Published by 12$|$TAG7
PUB$|$670$|$13$|$683$|$Object 683$|$Published by 13$|$TAG7
PUB$|$671$|$14$|$685$|$Object 685$|$Published by 14$|$TAG7
PUB$|$672$|$15$|$687$|$Object 687$|$Published by 15$|$TAG7
PUB$|$673$|$16$|$689$|$Object 689$|$Published by 16$|$TAG7
PUB$|$674$|$19$|$693$|$Object 693$|$Published by 19$|$TAG7
PUB$|$675$|$20$|$695$|$Object 695$|$Published by 20$|$TAG7
PUB$|$676$|$21$|$697$|$Object 697$|$Published by 21$|$TAG7
PUB$|$677$|$24$|$701$|$Object 701$|$Published by 24$|$TAG7
PUB$|$678$|$25$|$703$|$Object 703$|$Published by 25$|$TAG7
PUB$|$679$|$26$|$705$|$Object 705$|$Published by 26$|$TAG7
PUB$|$680$|$28$|$708$|$Object 708$|$Published by 28$|$TAG7
PUB$|$681$|$35$|$716$|$Object 716$|$Published by 35$|$TAG7
PUB$|$682$|$44$|$726$|$Object 726$|$Published by 44$|$TAG7
PUB$|$683$|$48$|$731$|$Object 731$|$Published by 48$|$TAG7
PUB$|$684$|$51$|$735$|$Object 735$|$Published by 51$|$TAG7
```

```
PUB$|$685$|$52$|$737$|$Object 737$|$Published by 52$|$TAG7
SUB_P$|$716$|$0$|$TAG0
SUB_P$|$717$|$0$|$TAG3
SUB_P$|$718$|$0$|$TAG5
SUB_P$|$719$|$0$|$TAG7
SUB_P$|$720$|$1$|$TAG4
SUB_P$|$721$|$1$|$TAG6
SUB_P$|$722$|$1$|$TAG7
SUB_P$|$723$|$2$|$TAG5
SUB_P$|$724$|$2$|$TAG6
SUB_P$|$725$|$2$|$TAG7
SUB_P$|$726$|$3$|$TAG0
SUB_P$|$727$|$3$|$TAG2
SUB_P$|$728$|$3$|$TAG4
SUB_P$|$729$|$3$|$TAG5
SUB_P$|$730$|$4$|$TAG0
SUB_P$|$731$|$4$|$TAG3
SUB_P$|$732$|$4$|$TAG6
SUB_P$|$733$|$5$|$TAG4
SUB_P$|$734$|$5$|$TAG6
SUB_P$|$735$|$6$|$TAG4
SUB_P$|$736$|$6$|$TAG5
SUB_P$|$737$|$6$|$TAG6
SUB_P$|$738$|$6$|$TAG7
SUB_P$|$739$|$7$|$TAG1
SUB_P$|$740$|$8$|$TAG0
SUB_P$|$741$|$8$|$TAG1
SUB_P$|$742$|$8$|$TAG4
SUB_P$|$743$|$8$|$TAG5
SUB_P$|$744$|$8$|$TAG7
SUB_P$|$745$|$9$|$TAG0
SUB_P$|$746$|$9$|$TAG1
SUB_P$|$747$|$9$|$TAG4
SUB_P$|$748$|$9$|$TAG5
SUB_P$|$749$|$10$|$TAG1
SUB_P$|$750$|$10$|$TAG4
SUB_P$|$751$|$10$|$TAG6
SUB_P$|$752$|$10$|$TAG7
SUB_P$|$753$|$11$|$TAG4
SUB_P$|$754$|$11$|$TAG6
SUB_P$|$755$|$12$|$TAG5
SUB_P$|$756$|$12$|$TAG7
SUB_P$|$757$|$13$|$TAG0
SUB_P$|$758$|$13$|$TAG2
SUB_P$|$759$|$13$|$TAG3
SUB_P$|$760$|$13$|$TAG6
SUB_P$|$761$|$13$|$TAG7
SUB_P$|$762$|$14$|$TAG1
SUB_P$|$763$|$14$|$TAG4
SUB_P$|$764$|$14$|$TAG5
```

```
SUB_P$|$765$|$15$|$TAG0
SUB_P$|$766$|$15$|$TAG6
SUB_P$|$767$|$16$|$TAG0
SUB_P$|$768$|$16$|$TAG1
SUB_P$|$769$|$16$|$TAG5
SUB_P$|$770$|$16$|$TAG7
SUB_P$|$771$|$17$|$TAG0
SUB_P$|$772$|$17$|$TAG2
SUB_P$|$773$|$17$|$TAG7
SUB_P$|$774$|$18$|$TAG0
SUB_P$|$775$|$18$|$TAG1
SUB_P$|$776$|$18$|$TAG3
SUB_P$|$777$|$18$|$TAG4
SUB_P$|$778$|$18$|$TAG7
SUB_P$|$779$|$20$|$TAG0
SUB_P$|$780$|$20$|$TAG1
SUB_P$|$781$|$20$|$TAG3
SUB_P$|$782$|$21$|$TAG4
SUB_P$|$783$|$21$|$TAG5
SUB_P$|$784$|$22$|$TAG2
SUB_P$|$785$|$22$|$TAG4
SUB_P$|$786$|$22$|$TAG5
SUB_P$|$787$|$22$|$TAG6
SUB_P$|$788$|$23$|$TAG1
SUB_P$|$789$|$23$|$TAG3
SUB_P$|$790$|$23$|$TAG4
SUB_P$|$791$|$23$|$TAG5
SUB_P$|$792$|$23$|$TAG7
SUB_P$|$793$|$24$|$TAG3
SUB_P$|$794$|$24$|$TAG7
SUB_P$|$795$|$25$|$TAG1
SUB_P$|$796$|$25$|$TAG3
SUB_P$|$797$|$25$|$TAG5
SUB_P$|$798$|$25$|$TAG7
SUB_P$|$799$|$26$|$TAG1
SUB_P$|$800$|$26$|$TAG3
SUB_P$|$801$|$26$|$TAG7
SUB_P$|$802$|$27$|$TAG0
SUB_P$|$803$|$27$|$TAG6
SUB_P$|$804$|$28$|$TAG0
SUB_P$|$805$|$28$|$TAG3
SUB_P$|$806$|$28$|$TAG5
SUB_P$|$807$|$28$|$TAG6
SUB_P$|$808$|$28$|$TAG7
SUB_P$|$809$|$29$|$TAG1
SUB_P$|$810$|$29$|$TAG7
SUB_P$|$811$|$30$|$TAG3
SUB_P$|$812$|$31$|$TAG2
SUB_P$|$813$|$31$|$TAG4
SUB_P$|$814$|$32$|$TAG1
```

```
SUB_P$|$815$|$32$|$TAG2
SUB_P$|$816$|$32$|$TAG3
SUB_P$|$817$|$32$|$TAG5
SUB_P$|$818$|$33$|$TAG3
SUB_P$|$819$|$33$|$TAG6
SUB_P$|$820$|$33$|$TAG7
SUB_P$|$821$|$34$|$TAG1
SUB_P$|$822$|$34$|$TAG5
SUB_P$|$823$|$34$|$TAG6
SUB_P$|$824$|$35$|$TAG1
SUB_P$|$825$|$35$|$TAG2
SUB_P$|$826$|$35$|$TAG3
SUB_P$|$827$|$35$|$TAG4
SUB_P$|$828$|$35$|$TAG5
SUB_P$|$829$|$36$|$TAG0
SUB_P$|$830$|$36$|$TAG7
SUB_P$|$831$|$37$|$TAG2
SUB_P$|$832$|$37$|$TAG3
SUB_P$|$833$|$37$|$TAG6
SUB_P$|$834$|$38$|$TAG5
SUB_P$|$835$|$38$|$TAG7
SUB_P$|$836$|$39$|$TAG1
SUB_P$|$837$|$40$|$TAG2
SUB_P$|$838$|$40$|$TAG4
SUB_P$|$839$|$40$|$TAG6
SUB_P$|$840$|$40$|$TAG7
SUB_P$|$841$|$41$|$TAG1
SUB_P$|$842$|$41$|$TAG2
SUB_P$|$843$|$41$|$TAG4
SUB_P$|$844$|$42$|$TAG2
SUB_P$|$845$|$43$|$TAG0
SUB_P$|$846$|$43$|$TAG2
SUB_P$|$847$|$44$|$TAG2
SUB_P$|$848$|$44$|$TAG3
SUB_P$|$849$|$44$|$TAG6
SUB_P$|$850$|$44$|$TAG7
SUB_P$|$851$|$45$|$TAG2
SUB_P$|$852$|$47$|$TAG2
SUB_P$|$853$|$47$|$TAG5
SUB_P$|$854$|$48$|$TAG0
SUB_P$|$855$|$48$|$TAG3
SUB_P$|$856$|$48$|$TAG4
SUB_P$|$857$|$48$|$TAG7
SUB_P$|$858$|$49$|$TAG1
SUB_P$|$859$|$49$|$TAG4
SUB_P$|$860$|$49$|$TAG7
SUB_P$|$861$|$51$|$TAG2
SUB_P$|$862$|$51$|$TAG4
SUB_P$|$863$|$51$|$TAG7
SUB_P$|$864$|$52$|$TAG0
```

```
SUB_P$|$865$|$52$|$TAG4
SUB_P$|$866$|$52$|$TAG7
SUB_P$|$867$|$53$|$TAG0
SUB_P$|$868$|$53$|$TAG1
SUB_P$|$869$|$53$|$TAG4
SUB_P$|$870$|$54$|$TAG3
SUB_P$|$871$|$54$|$TAG5
SUB_P$|$872$|$55$|$TAG3
SUB_P$|$873$|$56$|$TAG6
SUB_P$|$874$|$57$|$TAG0
SUB_P$|$875$|$57$|$TAG2
SUB_P$|$876$|$57$|$TAG7
SUB_P$|$877$|$58$|$TAG0
SUB_P$|$878$|$58$|$TAG3
SUB_P$|$879$|$58$|$TAG7
SUB_P$|$880$|$59$|$TAG0
SUB_P$|$881$|$59$|$TAG2
SUB_P$|$882$|$59$|$TAG4
SUB_P$|$883$|$59$|$TAG6
SUB_P$|$884$|$59$|$TAG7
SUB_P$|$885$|$60$|$TAG1
SUB_P$|$886$|$60$|$TAG2
SUB_P$|$887$|$60$|$TAG3
SUB_P$|$888$|$60$|$TAG7
SUB_P$|$889$|$61$|$TAG0
SUB_P$|$890$|$61$|$TAG2
SUB_P$|$891$|$61$|$TAG3
SUB_P$|$892$|$61$|$TAG6
SUB_P$|$893$|$61$|$TAG7
SUB_P$|$894$|$62$|$TAG2
SUB_P$|$895$|$62$|$TAG3
SUB_P$|$896$|$62$|$TAG7
SUB_P$|$897$|$63$|$TAG0
SUB_P$|$898$|$63$|$TAG2
SUB_P$|$899$|$63$|$TAG3
SUB_P$|$900$|$63$|$TAG4
SYNC$|$1021$|$
```

Annex II

System Reactivity Trace

```
NODE$|$0$|$0
NODE$|$1$|$1
NODE$|$2$|$2
NODE$|$3$|$3
NODE$|$4$|$4
NODE$|$5$|$5
NODE$|$6$|$6
NODE$|$7$|$7
NODE$|$8$|$8
NODE$|$9$|$9
NODE$|$10$|$10
NODE$|$11$|$11
SUB_F$|$42$|$0$|$TAG0
SUB_F$|$43$|$1$|$TAG0
SUB_F$|$44$|$2$|$TAG0
SUB_F$|$45$|$3$|$TAG0
SUB_F$|$46$|$4$|$TAG0
SUB_F$|$47$|$5$|$TAG0
SUB_F$|$48$|$6$|$TAG0
SUB_F$|$49$|$7$|$TAG0
SUB_F$|$50$|$8$|$TAG0
SUB_F$|$51$|$9$|$TAG0
SUB_F$|$52$|$10$|$TAG0
SUB_F$|$53$|$0$|$TAG1
SUB_F$|$54$|$1$|$TAG1
SUB_F$|$55$|$2$|$TAG1
SUB_F$|$56$|$3$|$TAG1
SUB_F$|$57$|$4$|$TAG1
SUB_F$|$58$|$5$|$TAG1
SUB_F$|$59$|$6$|$TAG1
SUB_F$|$60$|$7$|$TAG1
SUB_F$|$61$|$8$|$TAG1
SUB_F$|$62$|$9$|$TAG1
SUB_F$|$63$|$10$|$TAG1
SUB_F$|$64$|$0$|$TAG2
```

```
SUB_F$|$65$|$1$|$TAG2
SUB_F$|$66$|$2$|$TAG2
SUB_F$|$67$|$3$|$TAG2
SUB_F$|$68$|$4$|$TAG2
SUB_F$|$69$|$5$|$TAG2
SUB_F$|$70$|$6$|$TAG2
SUB_F$|$71$|$7$|$TAG2
SUB_F$|$72$|$8$|$TAG2
SUB_F$|$73$|$9$|$TAG2
SUB_F$|$74$|$10$|$TAG2
SUB_F$|$75$|$0$|$TAG3
SUB_F$|$76$|$1$|$TAG3
SUB_F$|$77$|$2$|$TAG3
SUB_F$|$78$|$3$|$TAG3
SUB_F$|$79$|$4$|$TAG3
SUB_F$|$80$|$5$|$TAG3
SUB_F$|$81$|$6$|$TAG3
SUB_F$|$82$|$7$|$TAG3
SUB_F$|$83$|$8$|$TAG3
SUB_F$|$84$|$9$|$TAG3
SUB_F$|$85$|$10$|$TAG3
SUB_F$|$86$|$0$|$TAG4
SUB_F$|$87$|$1$|$TAG4
SUB_F$|$88$|$2$|$TAG4
SUB_F$|$89$|$3$|$TAG4
SUB_F$|$90$|$4$|$TAG4
SUB_F$|$91$|$5$|$TAG4
SUB_F$|$92$|$6$|$TAG4
SUB_F$|$93$|$7$|$TAG4
SUB_F$|$94$|$8$|$TAG4
SUB_F$|$95$|$9$|$TAG4
SUB_F$|$96$|$10$|$TAG4
SUB_F$|$97$|$0$|$TAG5
SUB_F$|$98$|$1$|$TAG5
SUB_F$|$99$|$2$|$TAG5
SUB_F$|$100$|$3$|$TAG5
SUB_F$|$101$|$4$|$TAG5
SUB_F$|$102$|$5$|$TAG5
SUB_F$|$103$|$6$|$TAG5
SUB_F$|$104$|$7$|$TAG5
SUB_F$|$105$|$8$|$TAG5
SUB_F$|$106$|$9$|$TAG5
SUB_F$|$107$|$10$|$TAG5
SUB_F$|$108$|$0$|$TAG6
SUB_F$|$109$|$1$|$TAG6
SUB_F$|$110$|$2$|$TAG6
SUB_F$|$111$|$3$|$TAG6
SUB_F$|$112$|$4$|$TAG6
SUB_F$|$113$|$5$|$TAG6
SUB_F$|$114$|$6$|$TAG6
```

```
SUB_F$|$115$|$7$|$TAG6
SUB_F$|$116$|$8$|$TAG6
SUB_F$|$117$|$9$|$TAG6
SUB_F$|$118$|$10$|$TAG6
SUB_F$|$119$|$0$|$TAG7
SUB_F$|$120$|$1$|$TAG7
SUB_F$|$121$|$2$|$TAG7
SUB_F$|$122$|$3$|$TAG7
SUB_F$|$123$|$4$|$TAG7
SUB_F$|$124$|$5$|$TAG7
SUB_F$|$125$|$6$|$TAG7
SUB_F$|$126$|$7$|$TAG7
SUB_F$|$127$|$8$|$TAG7
SUB_F$|$128$|$9$|$TAG7
SUB_F$|$129$|$10$|$TAG7
SUB_F$|$130$|$0$|$TAG8
SUB_F$|$131$|$1$|$TAG8
SUB_F$|$132$|$2$|$TAG8
SUB_F$|$133$|$3$|$TAG8
SUB_F$|$134$|$4$|$TAG8
SUB_F$|$135$|$5$|$TAG8
SUB_F$|$136$|$6$|$TAG8
SUB_F$|$137$|$7$|$TAG8
SUB_F$|$138$|$8$|$TAG8
SUB_F$|$139$|$9$|$TAG8
SUB_F$|$140$|$10$|$TAG8
SUB_F$|$141$|$0$|$TAG9
SUB_F$|$142$|$1$|$TAG9
SUB_F$|$143$|$2$|$TAG9
SUB_F$|$144$|$3$|$TAG9
SUB_F$|$145$|$4$|$TAG9
SUB_F$|$146$|$5$|$TAG9
SUB_F$|$147$|$6$|$TAG9
SUB_F$|$148$|$7$|$TAG9
SUB_F$|$149$|$8$|$TAG9
SUB_F$|$150$|$9$|$TAG9
SUB_F$|$151$|$10$|$TAG9
SUB_F$|$152$|$0$|$TAG10
SUB_F$|$153$|$1$|$TAG10
SUB_F$|$154$|$2$|$TAG10
SUB_F$|$155$|$3$|$TAG10
SUB_F$|$156$|$4$|$TAG10
SUB_F$|$157$|$5$|$TAG10
SUB_F$|$158$|$6$|$TAG10
SUB_F$|$159$|$7$|$TAG10
SUB_F$|$160$|$8$|$TAG10
SUB_F$|$161$|$9$|$TAG10
SUB_F$|$162$|$10$|$TAG10
SUB_F$|$163$|$0$|$TAG11
SUB_F$|$164$|$1$|$TAG11
```

```
SUB_F$|$165$|$2$|$TAG11
SUB_F$|$166$|$3$|$TAG11
SUB_F$|$167$|$4$|$TAG11
SUB_F$|$168$|$5$|$TAG11
SUB_F$|$169$|$6$|$TAG11
SUB_F$|$170$|$7$|$TAG11
SUB_F$|$171$|$8$|$TAG11
SUB_F$|$172$|$9$|$TAG11
SUB_F$|$173$|$10$|$TAG11
SUB_F$|$174$|$11$|$TAG11
PUB$|$185$|$0$|$185$|$Object 185$|$Published by 0$|$TAG0
PUB$|$186$|$1$|$187$|$Object 187$|$Published by 1$|$TAG1
PUB$|$187$|$2$|$189$|$Object 189$|$Published by 2$|$TAG2
PUB$|$188$|$3$|$191$|$Object 191$|$Published by 3$|$TAG3
PUB$|$189$|$4$|$193$|$Object 193$|$Published by 4$|$TAG4
PUB$|$190$|$5$|$195$|$Object 195$|$Published by 5$|$TAG5
PUB$|$191$|$6$|$197$|$Object 197$|$Published by 6$|$TAG6
PUB$|$192$|$7$|$199$|$Object 199$|$Published by 7$|$TAG7
PUB$|$193$|$8$|$201$|$Object 201$|$Published by 8$|$TAG8
PUB$|$194$|$9$|$203$|$Object 203$|$Published by 9$|$TAG9
PUB$|$195$|$10$|$205$|$Object 205$|$Published by 10$|$TAG10
PUB$|$206$|$0$|$206$|$Object 206$|$Published by 0$|$TAG11
PUB$|$207$|$1$|$208$|$Object 208$|$Published by 1$|$TAG11
PUB$|$208$|$2$|$210$|$Object 210$|$Published by 2$|$TAG11
PUB$|$209$|$3$|$212$|$Object 212$|$Published by 3$|$TAG11
PUB$|$210$|$4$|$214$|$Object 214$|$Published by 4$|$TAG11
PUB$|$211$|$5$|$216$|$Object 216$|$Published by 5$|$TAG11
PUB$|$212$|$6$|$218$|$Object 218$|$Published by 6$|$TAG11
PUB$|$213$|$7$|$220$|$Object 220$|$Published by 7$|$TAG11
PUB$|$214$|$8$|$222$|$Object 222$|$Published by 8$|$TAG11
PUB$|$215$|$9$|$224$|$Object 224$|$Published by 9$|$TAG11
PUB$|$216$|$10$|$226$|$Object 226$|$Published by 10$|$TAG11
PUB$|$217$|$11$|$228$|$Object 228$|$Published by 11$|$TAG11
SUB_P$|$338$|$11$|$TAG0
SUB_P$|$349$|$11$|$TAG1
SUB_P$|$359$|$11$|$TAG2
SUB_P$|$368$|$11$|$TAG3
SUB_P$|$376$|$11$|$TAG4
SUB_P$|$383$|$11$|$TAG5
SUB_P$|$389$|$11$|$TAG6
SUB_P$|$394$|$11$|$TAG7
SUB_P$|$398$|$11$|$TAG8
SUB_P$|$401$|$11$|$TAG9
SUB_P$|$403$|$11$|$TAG10
SYNC$|$414$|$
```

Annex III

System Overhead Trace

```
NODE$|$0$|$0
NODE$|$1$|$1
NODE$|$2$|$2
NODE$|$3$|$3
NODE$|$4$|$4
NODE$|$5$|$5
NODE$|$6$|$6
NODE$|$7$|$7
NODE$|$8$|$8
NODE$|$9$|$9
NODE$|$10$|$10
NODE$|$11$|$11
NODE$|$12$|$12
NODE$|$13$|$13
NODE$|$14$|$14
NODE$|$15$|$15
SUB_F$|$46$|$1$|$TAG0
SUB_F$|$47$|$4$|$TAG0
SUB_F$|$48$|$6$|$TAG0
SUB_F$|$49$|$8$|$TAG0
SUB_F$|$50$|$9$|$TAG0
SUB_F$|$51$|$10$|$TAG0
SUB_F$|$52$|$12$|$TAG0
SUB_F$|$53$|$13$|$TAG0
SUB_F$|$54$|$15$|$TAG0
SUB_F$|$55$|$0$|$TAG1
SUB_F$|$56$|$3$|$TAG1
SUB_F$|$57$|$6$|$TAG1
SUB_F$|$58$|$8$|$TAG1
SUB_F$|$59$|$11$|$TAG1
SUB_F$|$60$|$13$|$TAG1
SUB_F$|$61$|$14$|$TAG1
SUB_F$|$62$|$2$|$TAG2
SUB_F$|$63$|$3$|$TAG2
SUB_F$|$64$|$4$|$TAG2
```

```
SUB_F$|$65$|$7$|$TAG2
SUB_F$|$66$|$8$|$TAG2
SUB_F$|$67$|$9$|$TAG2
SUB_F$|$68$|$12$|$TAG2
SUB_F$|$69$|$14$|$TAG2
SUB_F$|$70$|$15$|$TAG2
SUB_F$|$71$|$0$|$TAG3
SUB_F$|$72$|$3$|$TAG3
SUB_F$|$73$|$4$|$TAG3
SUB_F$|$74$|$5$|$TAG3
SUB_F$|$75$|$9$|$TAG3
SUB_F$|$76$|$11$|$TAG3
SUB_F$|$77$|$12$|$TAG3
SUB_F$|$78$|$13$|$TAG3
SUB_F$|$79$|$1$|$TAG4
SUB_F$|$80$|$4$|$TAG4
SUB_F$|$81$|$7$|$TAG4
SUB_F$|$82$|$9$|$TAG4
SUB_F$|$83$|$10$|$TAG4
SUB_F$|$84$|$12$|$TAG4
SUB_F$|$85$|$13$|$TAG4
SUB_F$|$86$|$15$|$TAG4
SUB_F$|$87$|$8$|$TAG5
SUB_F$|$88$|$10$|$TAG5
SUB_F$|$89$|$11$|$TAG5
SUB_F$|$90$|$12$|$TAG5
SUB_F$|$91$|$15$|$TAG5
SUB_F$|$92$|$1$|$TAG6
SUB_F$|$93$|$2$|$TAG6
SUB_F$|$94$|$3$|$TAG6
SUB_F$|$95$|$5$|$TAG6
SUB_F$|$96$|$6$|$TAG6
SUB_F$|$97$|$9$|$TAG6
SUB_F$|$98$|$10$|$TAG6
SUB_F$|$99$|$12$|$TAG6
SUB_F$|$100$|$14$|$TAG6
SUB_F$|$101$|$15$|$TAG6
SUB_F$|$102$|$0$|$TAG7
SUB_F$|$103$|$1$|$TAG7
SUB_F$|$104$|$3$|$TAG7
SUB_F$|$105$|$6$|$TAG7
SUB_F$|$106$|$8$|$TAG7
SUB_F$|$107$|$11$|$TAG7
SUB_F$|$108$|$12$|$TAG7
SUB_F$|$109$|$15$|$TAG7
PUB$|$110$|$0$|$110$|$Object 110$|$Published by 0$|$TAG0
PUB$|$111$|$1$|$112$|$Object 112$|$Published by 1$|$TAG0
PUB$|$112$|$3$|$115$|$Object 115$|$Published by 3$|$TAG0
PUB$|$113$|$6$|$119$|$Object 119$|$Published by 6$|$TAG0
PUB$|$114$|$9$|$123$|$Object 123$|$Published by 9$|$TAG0
```

```
PUB$|$115$|$10$|$125$|$Object 125$|$Published by 10$|$TAG0
PUB$|$116$|$12$|$128$|$Object 128$|$Published by 12$|$TAG0
PUB$|$117$|$13$|$130$|$Object 130$|$Published by 13$|$TAG0
PUB$|$118$|$4$|$122$|$Object 122$|$Published by 4$|$TAG1
PUB$|$119$|$7$|$126$|$Object 126$|$Published by 7$|$TAG1
PUB$|$120$|$12$|$132$|$Object 132$|$Published by 12$|$TAG1
PUB$|$121$|$13$|$134$|$Object 134$|$Published by 13$|$TAG1
PUB$|$122$|$0$|$122$|$Object 122$|$Published by 0$|$TAG2
PUB$|$123$|$2$|$125$|$Object 125$|$Published by 2$|$TAG2
PUB$|$124$|$3$|$127$|$Object 127$|$Published by 3$|$TAG2
PUB$|$125$|$4$|$129$|$Object 129$|$Published by 4$|$TAG2
PUB$|$126$|$7$|$133$|$Object 133$|$Published by 7$|$TAG2
PUB$|$127$|$9$|$136$|$Object 136$|$Published by 9$|$TAG2
PUB$|$128$|$10$|$138$|$Object 138$|$Published by 10$|$TAG2
PUB$|$129$|$11$|$140$|$Object 140$|$Published by 11$|$TAG2
PUB$|$130$|$12$|$142$|$Object 142$|$Published by 12$|$TAG2
PUB$|$131$|$0$|$131$|$Object 131$|$Published by 0$|$TAG3
PUB$|$132$|$2$|$134$|$Object 134$|$Published by 2$|$TAG3
PUB$|$133$|$3$|$136$|$Object 136$|$Published by 3$|$TAG3
PUB$|$134$|$4$|$138$|$Object 138$|$Published by 4$|$TAG3
PUB$|$135$|$6$|$141$|$Object 141$|$Published by 6$|$TAG3
PUB$|$136$|$7$|$143$|$Object 143$|$Published by 7$|$TAG3
PUB$|$137$|$8$|$145$|$Object 145$|$Published by 8$|$TAG3
PUB$|$138$|$10$|$148$|$Object 148$|$Published by 10$|$TAG3
PUB$|$139$|$11$|$150$|$Object 150$|$Published by 11$|$TAG3
PUB$|$140$|$14$|$154$|$Object 154$|$Published by 14$|$TAG3
PUB$|$141$|$15$|$156$|$Object 156$|$Published by 15$|$TAG3
PUB$|$142$|$3$|$145$|$Object 145$|$Published by 3$|$TAG4
PUB$|$143$|$8$|$151$|$Object 151$|$Published by 8$|$TAG4
PUB$|$144$|$11$|$155$|$Object 155$|$Published by 11$|$TAG4
PUB$|$145$|$13$|$158$|$Object 158$|$Published by 13$|$TAG4
PUB$|$146$|$15$|$161$|$Object 161$|$Published by 15$|$TAG4
PUB$|$147$|$2$|$149$|$Object 149$|$Published by 2$|$TAG5
PUB$|$148$|$3$|$151$|$Object 151$|$Published by 3$|$TAG5
PUB$|$149$|$4$|$153$|$Object 153$|$Published by 4$|$TAG5
PUB$|$150$|$6$|$156$|$Object 156$|$Published by 6$|$TAG5
PUB$|$151$|$8$|$159$|$Object 159$|$Published by 8$|$TAG5
PUB$|$152$|$9$|$161$|$Object 161$|$Published by 9$|$TAG5
PUB$|$153$|$11$|$164$|$Object 164$|$Published by 11$|$TAG5
PUB$|$154$|$12$|$166$|$Object 166$|$Published by 12$|$TAG5
PUB$|$155$|$15$|$170$|$Object 170$|$Published by 15$|$TAG5
PUB$|$156$|$0$|$156$|$Object 156$|$Published by 0$|$TAG6
PUB$|$157$|$2$|$159$|$Object 159$|$Published by 2$|$TAG6
PUB$|$158$|$5$|$163$|$Object 163$|$Published by 5$|$TAG6
PUB$|$159$|$6$|$165$|$Object 165$|$Published by 6$|$TAG6
PUB$|$160$|$11$|$171$|$Object 171$|$Published by 11$|$TAG6
PUB$|$161$|$12$|$173$|$Object 173$|$Published by 12$|$TAG6
PUB$|$162$|$15$|$177$|$Object 177$|$Published by 15$|$TAG6
PUB$|$163$|$0$|$163$|$Object 163$|$Published by 0$|$TAG7
PUB$|$164$|$1$|$165$|$Object 165$|$Published by 1$|$TAG7
```

```
PUB$|$165$|$2$|$167$|$Object 167$|$Published by 2$|$TAG7
PUB$|$166$|$3$|$169$|$Object 169$|$Published by 3$|$TAG7
PUB$|$167$|$4$|$171$|$Object 171$|$Published by 4$|$TAG7
PUB$|$168$|$5$|$173$|$Object 173$|$Published by 5$|$TAG7
PUB$|$169$|$8$|$177$|$Object 177$|$Published by 8$|$TAG7
PUB$|$170$|$9$|$179$|$Object 179$|$Published by 9$|$TAG7
PUB$|$171$|$12$|$183$|$Object 183$|$Published by 12$|$TAG7
PUB$|$172$|$13$|$185$|$Object 185$|$Published by 13$|$TAG7
PUB$|$173$|$14$|$187$|$Object 187$|$Published by 14$|$TAG7
SUB_P$|$174$|$0$|$TAG0
SUB_P$|$175$|$0$|$TAG2
SUB_P$|$176$|$0$|$TAG4
SUB_P$|$177$|$1$|$TAG2
SUB_P$|$178$|$1$|$TAG3
SUB_P$|$179$|$2$|$TAG0
SUB_P$|$180$|$2$|$TAG1
SUB_P$|$181$|$2$|$TAG3
SUB_P$|$182$|$2$|$TAG4
SUB_P$|$183$|$2$|$TAG5
SUB_P$|$184$|$3$|$TAG0
SUB_P$|$185$|$3$|$TAG5
SUB_P$|$186$|$4$|$TAG1
SUB_P$|$187$|$4$|$TAG5
SUB_P$|$188$|$4$|$TAG6
SUB_P$|$189$|$4$|$TAG7
SUB_P$|$190$|$5$|$TAG1
SUB_P$|$191$|$5$|$TAG2
SUB_P$|$192$|$5$|$TAG4
SUB_P$|$193$|$5$|$TAG5
SUB_P$|$194$|$6$|$TAG2
SUB_P$|$195$|$6$|$TAG3
SUB_P$|$196$|$6$|$TAG5
SUB_P$|$197$|$7$|$TAG0
SUB_P$|$198$|$7$|$TAG1
SUB_P$|$199$|$7$|$TAG3
SUB_P$|$200$|$7$|$TAG5
SUB_P$|$201$|$7$|$TAG6
SUB_P$|$202$|$7$|$TAG7
SUB_P$|$203$|$8$|$TAG3
SUB_P$|$204$|$8$|$TAG4
SUB_P$|$205$|$8$|$TAG6
SUB_P$|$206$|$9$|$TAG1
SUB_P$|$207$|$10$|$TAG1
SUB_P$|$208$|$10$|$TAG2
SUB_P$|$209$|$10$|$TAG3
SUB_P$|$210$|$10$|$TAG7
SUB_P$|$211$|$11$|$TAG0
SUB_P$|$212$|$11$|$TAG2
SUB_P$|$213$|$11$|$TAG4
SUB_P$|$214$|$12$|$TAG1
```

```
SUB_P$|$215$|$13$|$TAG2
SUB_P$|$216$|$13$|$TAG5
SUB_P$|$217$|$13$|$TAG6
SUB_P$|$218$|$13$|$TAG7
SUB_P$|$219$|$14$|$TAG3
SUB_P$|$220$|$14$|$TAG4
SUB_P$|$221$|$14$|$TAG7
SUB_P$|$222$|$15$|$TAG1
SUB_P$|$223$|$15$|$TAG3
SYNC$|$284$|$
```