ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Triennale in Ingegneria e Scienze Informatiche

Multi-platform distributed systems with Aggregate Computing in Kotlin: the case of proximity messaging

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Pianini Danilo

Candidato
Luca Marchi

Correlatore
Dott.ssa Cortecchia Angela

II Sessione di Laurea
Anno Accademico 2024-2025

Abstract

To my family.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

This chapter introduces the overall purpose and scope of the thesis. It begins by presenting the scientific and technological context in which this work is situated, highlighting the state of the art and the motivations that led to the development of a decentralized proximity-based messaging system. Subsequently, it outlines the research objectives, methodological approach, and main contributions of the study.

1.1 Context

In a world where Internet of Things (IoT) devices are becoming increasingly prevalent, the need for efficient and reliable communication systems is paramount (Figure 1.1).

The interactions between neighboring devices play a crucial role in enabling seamless data exchange and coordination, so there is the need to design networks with infrastructures that support scalability, adaptability and reusability [BPV15]. In the past, it was reasonable to use a programming model that focused on the individual computing device, and its relationship with one or more users. However, as systems have grown in scale and complexity with the number of computing devices rising, this method has become inadequate.

Traditional network architectures, rely heavily on centralized infrastructures, making them unsuitable for scenarios such as disaster recovery or interactions with

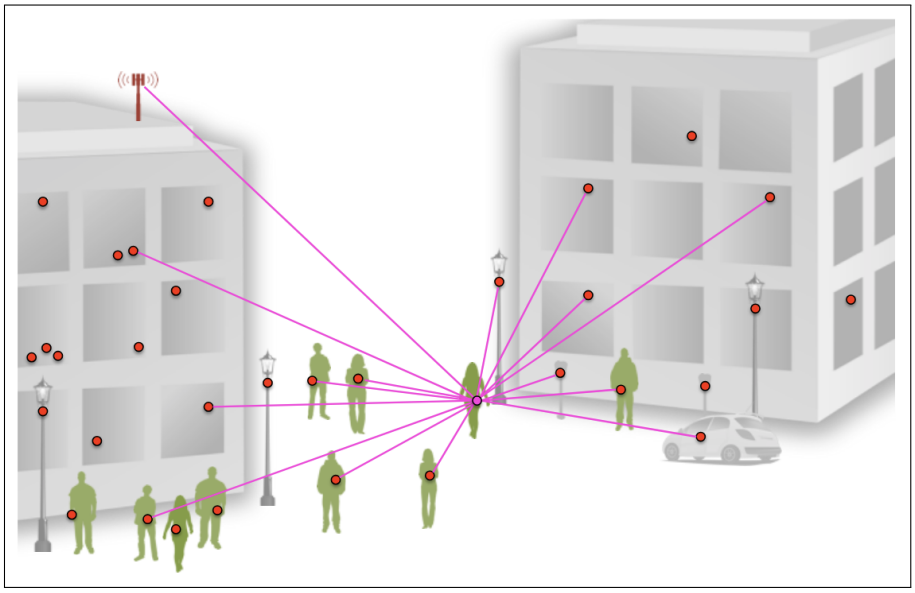

Figure 1.1: Our world is increasingly populated with a wide range of computing devices, embedded in our environment and with many opportunities for local and even location-independent interactions on fixed network infrastructures [BPV15].

neighboring devices. The computational model of *Aggregate Computing* provides a promising approach to address these challenges by enabling decentralized and self-organizing systems [VBD+19].

Building on this concept, this thesis explores how Aggregate Programming can support a proximity and decentralized messaging system in a network.

1.1.1 Aggregate Programming

Aggregate Programming is a distributed systems paradigm that simplifies programming large networks of devices by focusing on the global, system-level behavior rather than the individual behavior of each device. It shifts the abstraction to view the network as an aggregate, an agglomeration of nodes that collectively exhibit certain behaviors. This model is particularly well-suited for scenarios where devices need to coordinate and collaborate based on local interactions, such as in IoT applications.

As discussed in distributed field programming [VBD+19], there are principles to follow when designing large-scale systems:

- The mechanisms ensuring robust coordination should operate transparently in the background, so that programmers don't need to manage them directly.
- The composition of modules and subsystems should be simple and predictable, allowing developers to clearly understand the consequences of combining components.
- Large-scale distributed systems often consist of multiple heterogeneous subsystems, each of which may require different coordination strategies depending on the region or context in time.

Aggregate Programming aims to overcome these challenges through three fundamental principles:

1. The computational target is conceived as a region of the environment, with the underlying device-level details abstracted away, potentially even modeled as a continuous spatial domain.
2. The program logic is expressed as the manipulation of data structures that extend spatially and temporally across that region.
3. These computations are executed locally by individual devices within the region, which coordinate through resilient mechanisms and proximity-based interactions.

The foundation of this paradigm lies in *field calculus* [BPV15], a core set of constructs modeling device behavior and interaction. These essential features are captured in a tiny universal language suitable for mathematical analysis. The concept of **field** plays a central role in this context and originates from physics, where it is defined as a function mapping every point in a space-time domain to a scalar value.

According to [VBD+19], the *field value* ϕ is a function

$$\phi : D \to L \tag{1.1}$$

that maps each device δ in the domain D to a local value ℓ in the range L. This value can change in time with a *field evolution* that maps each point in

time to a field value. Figure 1.2 shows how Aggregate Programming abstracts the complexity of the underlying distributed network and its coordination challenges through a sequence of hierarchical abstraction layers.

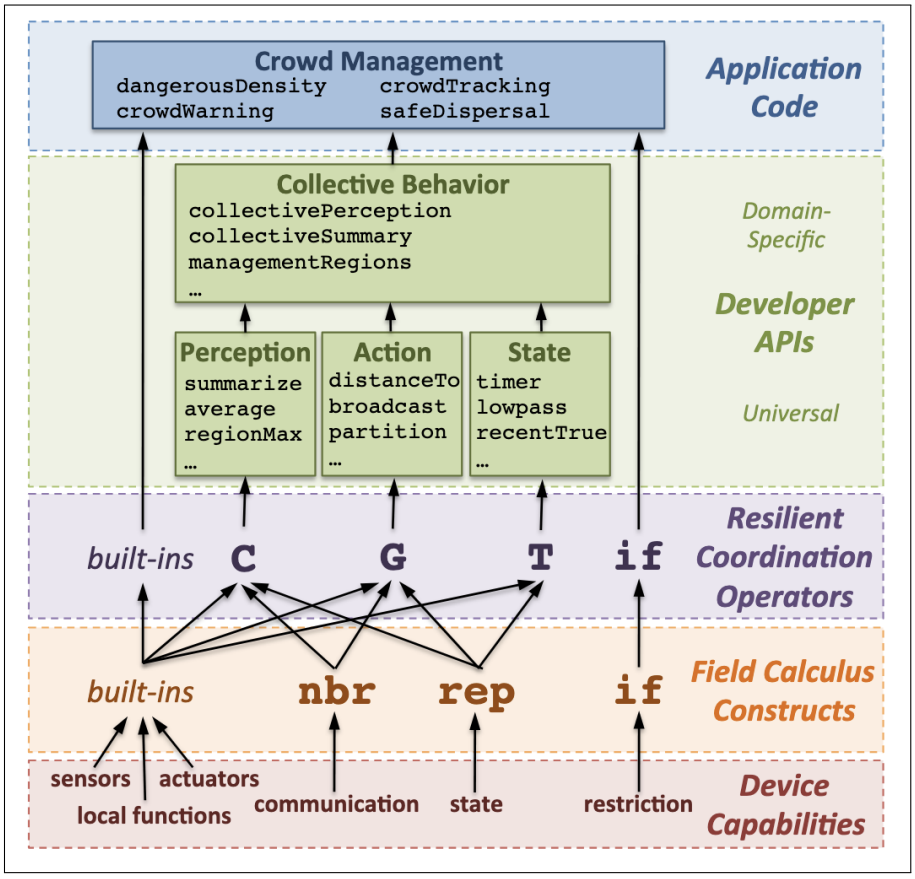

Figure 1.2: Layered structure of Aggregate Programming, illustrating the abstraction at each level [BPV15].

The messaging system developed in this thesis was implemented using *Collektive* [Cor24], an open-source framework designed to simplify the development of distributed systems through the principles of Aggregate Programming. Collektive provides developer-facing APIs (as in Figure 1.2) used to define and execute aggregate applications.

1.1.2 Collektive

Collektive is a framework with the purpose of simplifying the definition of Aggregate Computing systems. To achieve this, Collektive adopts the Field Calculus model, providing an intuitive way to define aggregate behavior without low-level coding. Built with Kotlin Multiplatform (KMP), it can be executed seamlessly on different targets. Moreover, a compiler plugin was introduced to support function alignment, automatically annotating aligned functions to ensure consistent coordination among nodes [Cor24].

The main features of this framework can be summarized as follows [Cor25]:

- **High-Level Abstraction**: Collektive streamlines the development of distributed behaviors by enabling developers to define coordination logic succinctly, without explicitly handling individual device states.
- **Declarative Programming**: The framework adopts a declarative paradigm, allowing developers to specify the desired system outcomes, while the underlying runtime manages the execution details.
- **Dynamic Adaptation**: Applications developed with Collektive can automatically adapt to network variations and external inputs, maintaining robustness in dynamic and unpredictable environments.

It provides a minimal Domain Specific Language (DSL) where the developer can specify the collective behaviour of a network of devices and the devices can directly communicate with each other and execute the same program.

1.2 Motivations

In this section motivations for implementing a decentralized proximity-based messaging system will be illustrated.

In recent years, the proliferation of IoT and mobile devices has increased the need for local communication systems capable of operating even without stable network infrastructure. Centralized architectures, although efficient in connected environments, fail to guarantee reliability and resilience in dynamic or disconnected scenarios. As devices are becoming pervasive and ubiquitous, new models

are required to enable coordination and information exchange without relying on fixed infrastructures. This need motivates the exploration of decentralized and distributed approaches, which can operate robustly in dynamic and connectivity-limited environments.

1.2.1 Problem statement

Traditional messaging systems and architectures often rely heavily on centralized servers and infrastructures, which are unsuitable and vulnerable for scenarios where connectivity is not guaranteed. These situations include **disaster recovery**, the process for restoring an organization's network infrastructure and operations after a disruptive event, such as a natural disaster, cyberattack, or system failure.

In such environments there are several vulnerabilities and challenges to consider:

- **Single point of failure**: Centralized systems are vulnerable to failures of the authority that makes decisions. This can lead to widespread service disruption and outages.
- **Scalability issues**: As the number of devices increases, centralized systems may struggle to manage the growing volume of data and communication, leading to bottlenecks and delayed responses.
- **Limited resilience**: Centralized architectures often lack the redundancy and fault tolerance needed to maintain functionality in the face of network disruptions or device failures.

The networks that need to overcome the issues above can benefit from Aggregate Computing principles, which provides a distributed architecture where the nodes can directly communicate with the neighbors.

1.2.2 Real world use cases

As mentioned in the previous sections, there are several scenarios where a decentralized proximity-based messaging system can be particularly beneficial.

The main use case of such system is in emergency situations. When natural disasters, or infrastructure failures occur, cellular networks and Internet connections often go down, making it difficult for people to communicate and coordinate rescue efforts. Nearby devices can form ad-hoc Bluetooth or Wi-Fi Direct networks, allowing users in the same area to propagate and share messages without relying on centralized servers.

This architecture can also be useful in large IoT networks like sensors systems or drones swarms, operating in remote areas and with the need to communicate without continuous Internet access.

The system's decentralized nature fits perfectly for:

- Exchanging local environmental data (temperature, humidity, motion).
- Maintaining distributed consensus or context awareness among nearby nodes.
- Avoiding single points of failure.

A practical example includes a network of environmental sensors deployed in a forest to monitor conditions and detect wildfires (Figure 1.3), or sensors in a smart agriculture field that share humidity data to coordinate irrigation locally.

1.2.3 Goal

This thesis can be divided into two main goals:

1. **Messaging algorithm**: The first goal is to design and implement a decentralized, proximity-based messaging algorithm grounded in the principles of Aggregate Computing. The network operates in a fully distributed manner, without any central server managing message exchange. Each node acts autonomously, coordinating with its neighbors through local interactions and collectively achieving self-organization, thereby eliminating single points of failure. Communication is proximity-based: devices can only interact with nearby nodes, and messages are propagated progressively across the network, layer by layer. The algorithm must also handle dynamic network topologies, where devices can join or leave the network at any time, ensuring robustness and adaptability. Message sources can define both an expiration time and

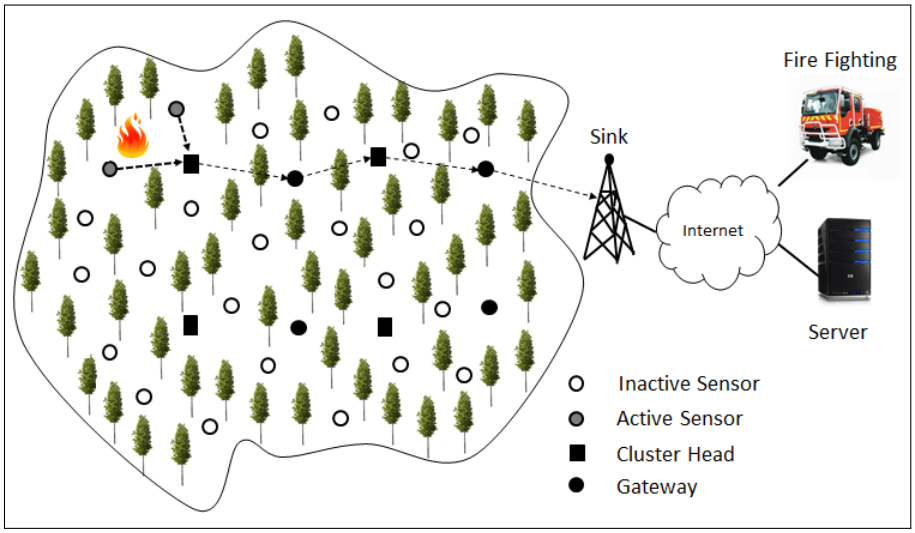

Figure 1.3: Real-world use case of a decentralized proximity-based messaging application built on Aggregate Computing principles. In this scenario, sensors located near the fire detect the event and propagate an alert message to neighboring nodes within the network [SBEK16].

a maximum propagation distance, enabling controlled diffusion and limiting unnecessary communication overhead. The system is implemented using the *Collektive* framework [Cor24], leveraging KMP to ensure cross-platform compatibility across different targets including JVM, Javascript (browser), Android, iOS, and native versions (for Windows, MacOS, and Linux, both for x86 and ARM CPUs).

2. **Mobile application**: The second goal is to develop a multiplatform mobile application using KMP. This application leverages the messaging algorithm as a core library to send and receive messages from devices in the network. The app must support both Android and iOS platforms providing a simple, user-friendly interface. The User Interface (UI) allows users to compose messages, configure parameters (expiration time and propagation distance), and view received messages in real-time. Another feature is the anonymization of the users, there is not any registration or login system, the users are completely anonymous. Anybody can use the app without providing any personal information.

1.3 State of Art

In this section, it will be presented the state of the art in the field of decentralized communication systems. The focus will be on the main existing systems, Aggregate Computing frameworks, and lightweight communication protocols.

1.3.1 Existing systems

Several communication technologies and systems have been developed over the years to facilitate decentralized data exchange among devices. The software that has spread the most and gained significant attention includes: *Briar*, *Bridgefy*, *FireChat*.

Briar

Briar [RSG+11] is an open-source Peer-to-Peer (P2P) messaging platform designed for secure, resilient communication without centralized servers. It supports multiple transport methods including Bluetooth, Wi-Fi and Tor, with all communication end-to-end encrypted. Briar provide a mailbox system *Briar Mailbox* that lets the users receive encrypted messages from their contacts while Briar is offline. Next time Briar comes online it will automatically fetch the messages from the Mailbox. The primary target audience includes activists, journalists, and civil society members who require secure communication in restricted or infrastructure-limited environments.

A notable feature is the ability to install Briar on an Android device directly from another device that already has it installed, facilitating deployment in disconnected scenarios. Nowadays the application is available only for Android devices, with no official iOS version.

Bridgefy

Bridgefy [Inc14] is a mobile messaging application that enables offline communication through Bluetooth mesh networking technology.

The platform allows smartphones to connect directly with nearby devices, creating a decentralized mesh network that can relay messages across multiple hops

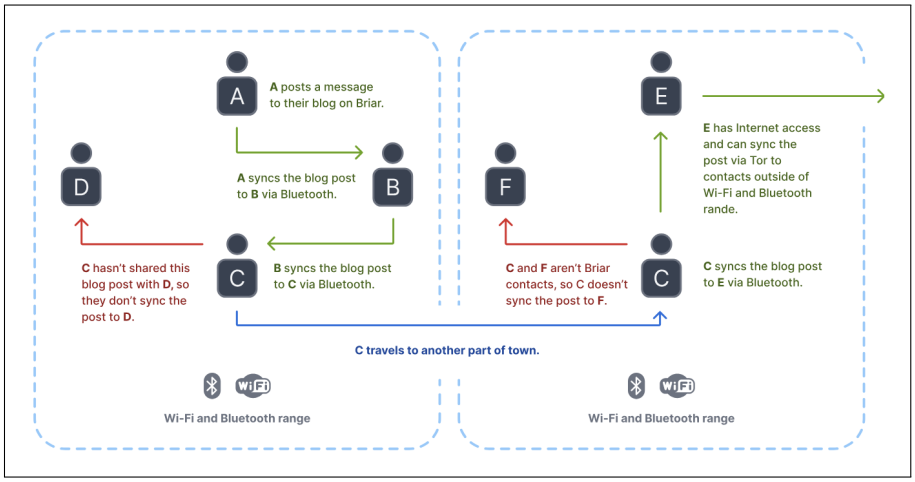

Figure 1.4: Diagram illustrating how Briar supports private messaging, public forums, and blogs through multiple transport methods such as Bluetooth, Wi-Fi, and the Internet. [RSG+11].

without requiring Internet connectivity. Originally designed for emergency situations and large gatherings where cellular networks become congested, Bridgefy has been deployed in various crisis scenarios including natural disasters and political protests.

Messages are automatically routed through intermediate devices, extending the effective communication range beyond direct Bluetooth connectivity (typically 100 meters). Bridgefy uses encryption protocols to secure message transmission; this capability was not available in the app's initial versions.

The service operates on both Android and iOS devices, with a user-friendly interface designed for rapid adoption during emergency situations. One limitation is the reliance on device density: the mesh network requires sufficient nearby users to establish reliable multi-hop communication paths.

FireChat

Firechat was the first popular example of a messaging application that used mesh network topology to communicate between users.

[MTN+17] FireChat relays messages from one device to another through each user's WiFi or Bluetooth radios, thereby creating a "mesh" of phones receiving

and relaying messages even when each message is not viewable on every device (Figure 1.5). This method of communication eliminates the need for centralized communication channels such as cellphone towers that every transmission must pass through. While this system is beneficial in overcrowded networks, it eliminates the possibility of controlling or monitoring the network at one point.

Although not originally intended for this purpose, FireChat was widely adopted during civil protests. The application gained popularity in Iraq in 2014, when government restrictions limited Internet access, and later during the 2014 Hong Kong protests (known as the Umbrella Revolution)

Nowadays Firechat is discontinued and no longer available on app stores, the official website is also offline.

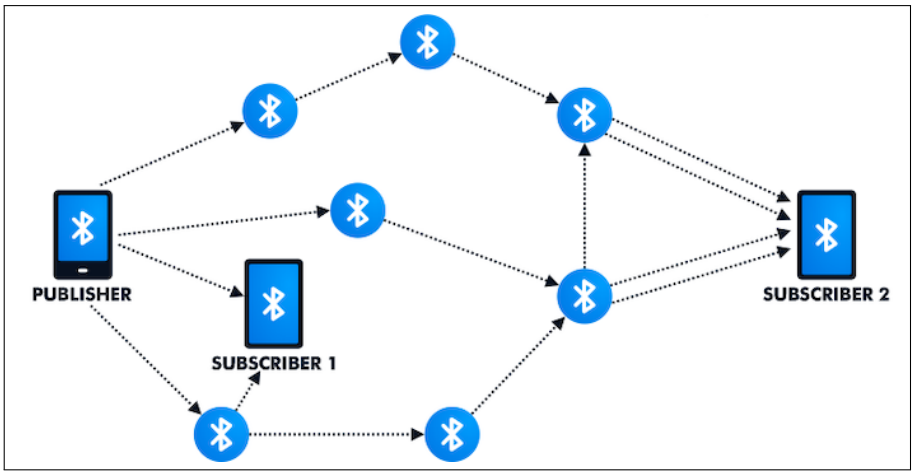

Figure 1.5: FireChat's mesh networking architecture, where messages are relayed through nearby devices using WiFi and Bluetooth, enabling communication without centralized infrastructure.

Final considerations

Those three illustrated applications have more or less the same purpose of providing a decentralized approach to messaging. *Briar* stands out for its strong emphasis on security and privacy, making it suitable for users in sensitive situations. *Bridgefy* is the most user-friendly and practical, focused on connecting users in emergency scenarios; encryption was included in later versions. *FireChat*

was the pioneering first popular example, but now outdated and discontinued. It inspired newer applications.

These applications demonstrate the feasibility of decentralized communication, they rely on predefined routing or broadcasting protocols. None of them incorporate *spatial awareness* or *self-organizing behavior*, features that are naturally modeled through *Aggregate Computing*.

1.3.2 Aggregate Computing frameworks

There are several frameworks and libraries that facilitate the development of Aggregate Computing applications, implementing the principles of field calculus and providing abstractions for defining collective behaviors. Some of the most notable

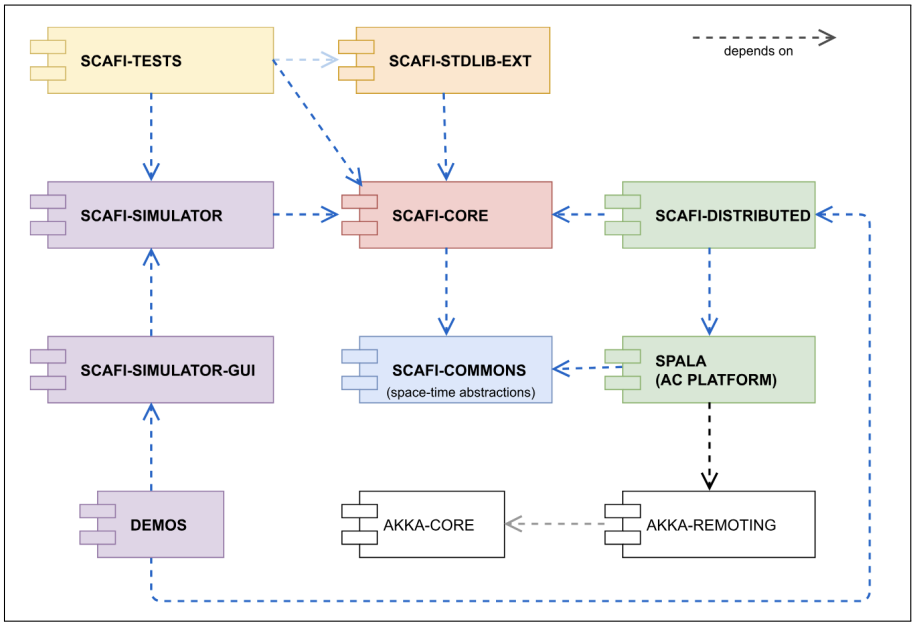

Figure 1.6: High-level architecture of the ScaFi framework, illustrating its components [CVAP22].

ones include:

- **ScaFi** (Scala Field) is a framework written in Scala for Aggregate Computing [CVAP22]. It provides a DSL and API modules for writing, testing, and running aggregate programs, the high-level architecture of ScaFi is depicted

in Figure 1.6. *ScaFi* also includes a simulation environment with a Graphical User Interface (GUI), integrated with the Alchemist Simulator [PMV13], an open-source software for simulating and experimenting with distributed systems.

The main applications are in the development of swarm systems, crowd management, wireless sensor network and smart city applications.

ScaFi only supports JVM and web-based applications.

- **Protelis** [PVB15] is a programming language designed to simplify the development of resilient and well-structured networked systems, and can be integrated into various simulation frameworks. It provides a Java-hosted implementation of the field calculus through the `protelis-lang` library, offering a domain-general foundational API for Aggregate Programming [FPBV17]. The Protelis architecture (Figure 1.7) relies on an interpreter that periodically executes pre-parsed aggregate programs, enabling communication among devices and interaction with the environment. However, since it is built on the Xtext framework for domain-specific languages and requires a JVM to run, its deployment is limited to Java-compatible devices, reducing platform heterogeneity.

1.3.3 Lightweight communication protocols

In the context of decentralized and distributed messaging systems, **lightweight communication protocols** are essential for efficient data exchange among devices with limited resources. A lightweight protocol minimizes overhead transmitted on top of the functional data, useful in scenarios where bandwidth, power, or processing capacity is limited. With the proliferation of IoT and mobile devices, these protocols have become crucial in modern networking.

Keeping in mind the distributed nature of the system, we can consider the following technologies.

Bluetooth Low Energy (BLE) formerly marked as *Bluetooth Smart*, is a wireless Personal Area Network (PAN) technology designed for short-range communication with low power consumption. BLE uses the same 2.4 GHz radio fre-

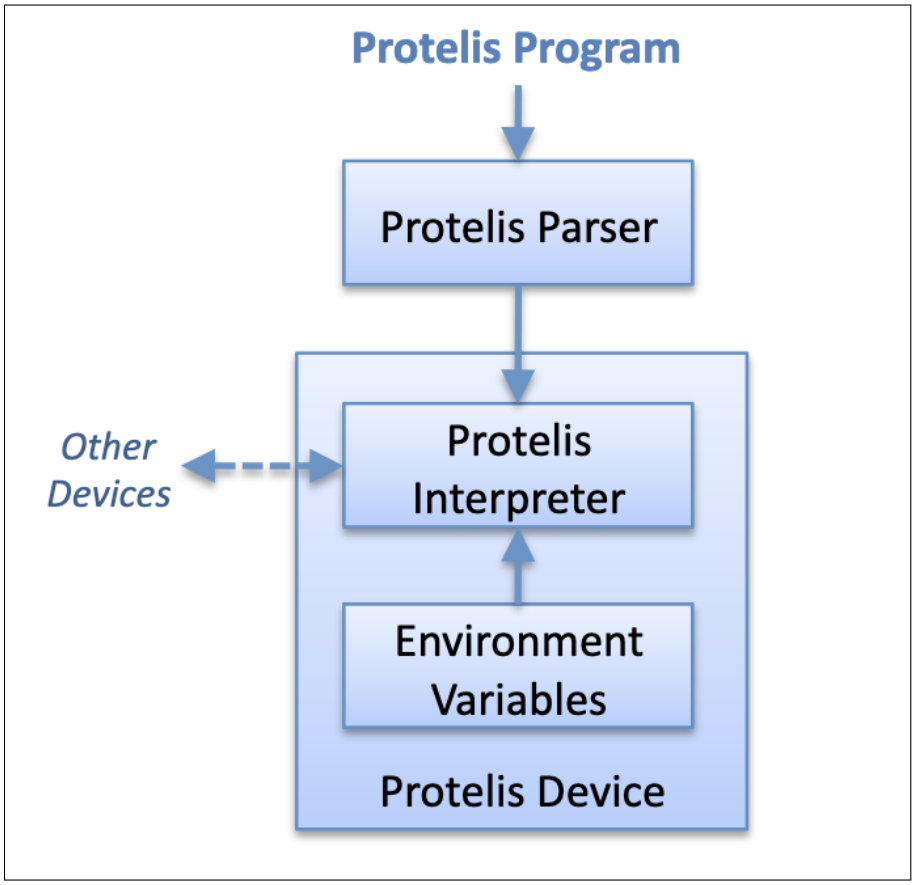

Figure 1.7: Protelis architecture.

quency as classic Bluetooth but employs a different protocol stack optimized for energy usage. It has no compatibility with classic Bluetooth, meaning that devices supporting only one of the two standards cannot communicate directly.

BLE supports device discovery and connection through low-duty advertising and scanning cycles, enabling the creation of ad-hoc networks for proximity-based communication.

Another similar protocol is **Wi-Fi Direct**, a wireless standard that enables devices to connect directly (using Wi-Fi) to each other without the need for a traditional access point or router. It allows P2P communication using standard Wi-Fi technology, offering higher data transfer rates and longer range compared to Bluetooth-based solutions.

The table Table 1.1 summarizes the main differences between BLE and Wi-Fi

Table 1.1: Comparison between BLE and Wi-Fi Direct.

Feature	**BLE**	**Wi-Fi Direct**
Communication range	Typically up to 50 meters (depending on environment)	Up to 200 meters (depending on device and power)
Data rate	Up to 2 Mbps (BLE 5.0)	Up to 250 Mbps
Consumption	Very low, optimized for battery-powered devices	Higher, comparable to standard Wi-Fi
Network topology	Simple connections or small mesh networks	Group-based topology with a dynamically assigned group owner
Connection setup	Fast, lightweight pairing through advertising and scanning	Slower setup, requires negotiation between devices
Use cases	Sensor networks, wearables, proximity-based messaging	File sharing, multimedia streaming, ad-hoc data exchange
Advantages	Energy efficient, widely supported, ideal for IoT	High throughput, longer range, infrastructure-free
Limitations	Limited bandwidth and range	Higher energy consumption, more complex setup

Direct.

Message Queuing Telemetry Transport (MQTT)

MQTT (Message Queuing Telemetry Transport) is a lightweight communication protocol based on a centralized *publish-subscribe* architecture. It relies on a central server, known as the *broker*, which manages message distribution among connected clients (Figure 1.8).

In this model, clients can either *publish* messages to specific *topics* or *subscribe* to topics of interest, receiving only data relevant to them [Usm21]. The broker acts as an intermediary that decouples message producers and consumers, ensuring

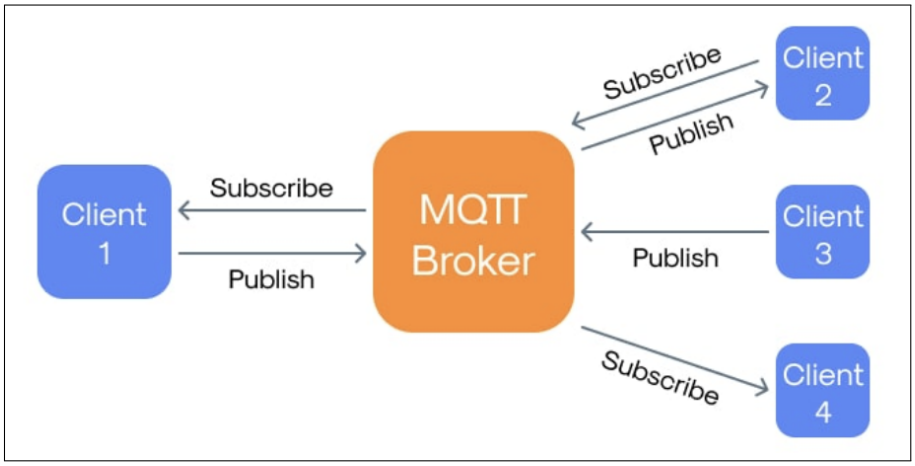

Figure 1.8: MQTT architecture illustrating the publish-subscribe model with clients and a central broker.

reliable and asynchronous communication.

MQTT is widely adopted in IoT systems due to its minimal packet overhead, low bandwidth consumption, and persistent session management, making it particularly suitable for constrained devices and unstable network conditions.

The protocol defines three levels of Quality of Service (QoS), which determine the reliability of message delivery:

- **QoS 0 - At most once:** messages are delivered according to the best effort of the underlying network, with no confirmation or retry mechanism. There is no acknowledgment from the receiver.
- **QoS 1 - At least once:** ensures that messages are delivered to the receiver at least one time, though duplicates may occur. It requires a PUBACK acknowledgment.
- **QoS 2 - Exactly once:** guarantees that each message is received only once by the intended recipient, at the cost of additional network overhead using using a four-step handshake (PUBLISH, PUBREC, PUBREL, PUBCOMP).

In the current implementation of the proposed application (Section 2), MQTT is employed as the communication layer to exchange messages across devices. Although it depends on a centralized broker and is therefore not fully decentralized,

its simplicity and efficiency make it an ideal solution for prototyping and testing the distributed messaging algorithm (Section 1).

Future developments may involve replacing MQTT with peer-to-peer (P2P) communication technologies, such as Bluetooth, to achieve full decentralization while maintaining the same messaging semantics.

Chapter 2

Architecture and Implementation

This chapter presents the main contributions of the thesis, with particular emphasis on the design and implementation of the decentralized messaging system. It details the architectural choices, software components, and underlying mechanisms that enable message propagation through gossip-based aggregate behaviors, highlighting how these elements collectively realize the system's objectives and support its cross-platform deployment.

2.1 Requirements

This section outlines both the functional and non-functional requirements of the messaging system developed in this thesis. It describes what the system is expected to accomplish, its qualities, constraints, and performances.

The requirements refer to both the messaging algorithm itself (see 1) and the mobile application (see 2).

2.1.1 Functional requirements

The functional requirements define the features, behaviors, and operations that the messaging system must support.

Aggregate Computing algorithm

The messaging algorithm represents the Aggregate Computing layer, the core logic of the system.

1. **Message propagation**: The algorithm must enable nodes to create and send messages to other nodes within the network.
2. **Decentralized and proximity-based communication**: Each node should be able to communicate only with its immediate neighbors, the message is propagated in the network through local interactions. There is no central authority managing the communication, the system must be fully distributed and autonomous.
3. **Parameter configurability**: The algorithm should allow message sources to define the expiration time of the message and the maximum propagation distance. With these parameters it is possible to control how long a message remains valid and how far it can spread in the network, limiting unnecessary communication overhead.
4. **Dynamic topology handling**: The network topology can change dynamically, with nodes joining or leaving the network at any time. The algorithm must be robust and adaptable to these changes, ensuring continuous message propagation without disruptions.
5. **Simulation support**: The algorithm must be executable within a simulated environment to facilitate testing and validation. This approach enables controlled experimentation under various network conditions and scenarios, ensuring the correctness and stability of the algorithm before its deployment on real devices.

Mobile Application

The mobile application serves as the UI for the messaging system, providing users both an interface and a runtime environment for the algorithm.

1. **Cross-platform compatibility**: The application must be supported on both Android and iOS platforms.

2. **User interface**: The application should provide a simple and intuitive UI that allows users to compose messages, set parameters (expiration time and propagation distance), and view received messages in real-time.

3. **GPS integration**: The application uses the device's GPS to determine its location, which is essential to compute distances between nodes for message propagation.

4. **Communication layer**: A lightweight communication protocol must be used to send messages between the various nodes in the network.

2.1.2 Non-functional requirements

The non-functional requirements specify the quality attributes, constraints, and performance criteria that the messaging system must meet, defining how well the functions should be performed.

One of the main non-functional requirements of the system is the **code alignment** between the simulator and the mobile application. This ensures that the same algorithmic logic is shared across both environments, minimizing discrepancies between simulation and real-world execution. In practice, the aggregate algorithm is implemented as a common API, which can be invoked identically within the simulation framework and the mobile runtime. This alignment is fundamental, as the distributed nature of the system requires extensive validation under multiple-node scenarios.

Another key non-functional requirement is **scalability**. The system must sustain acceptable performance levels as the number of participating nodes increases, ensuring that the aggregate computation cycle executes periodically without noticeable degradation on mobile devices.

In addition, the codebase must exhibit high **maintainability** and **extensibility**, enabling future developers to easily understand, modify, and enhance the system. These qualities are achieved through modular architectural design, comprehensive documentation, and adherence to established coding standards. Furthermore, the system architecture should be flexible enough to support the integration of new communication protocols or additional functionalities without

requiring significant refactoring.

2.1.3 Analysis

To address these requirements and challenges, the system adopts a **decentralized architecture** through the principles of **Aggregate Computing**. Each device is treated as an independent computational entity (or node) capable of locally executing aggregate functions and exchanging information only with its neighbors. This model allows global behaviors, such as message diffusion or decay, to emerge from simple local interactions. Message propagation is achieved through **gradient algorithm** based on **gossip**, which naturally reflect spatial proximity and temporal dynamics within the network.

From an implementation perspective, the *Collektive* framework is employed to define and simulate the aggregate behavior of the system, ensuring correctness and reproducibility under controlled conditions.

The communication layer is currently based on the **MQTT** protocol, which provides a lightweight and reliable publish/subscribe mechanism suitable for distributed messaging and initial prototyping.

Finally, the use of **KMP** enables a unified codebase that targets multiple platforms (Android, iOS, JVM) facilitating consistency between the simulated and real execution environments.

2.2 Network Architecture

In the following sections, the overall **architecture** of the messaging system will be described (Figure 2.1), focusing on both the design and communication model.

2.2.1 Design

The network is modeled as an undirected graph $G = (V, E)$, where V is the set of nodes representing individual devices, and E is the set of edges representing communication links between neighboring nodes (Figure 2.2). A **node** in the

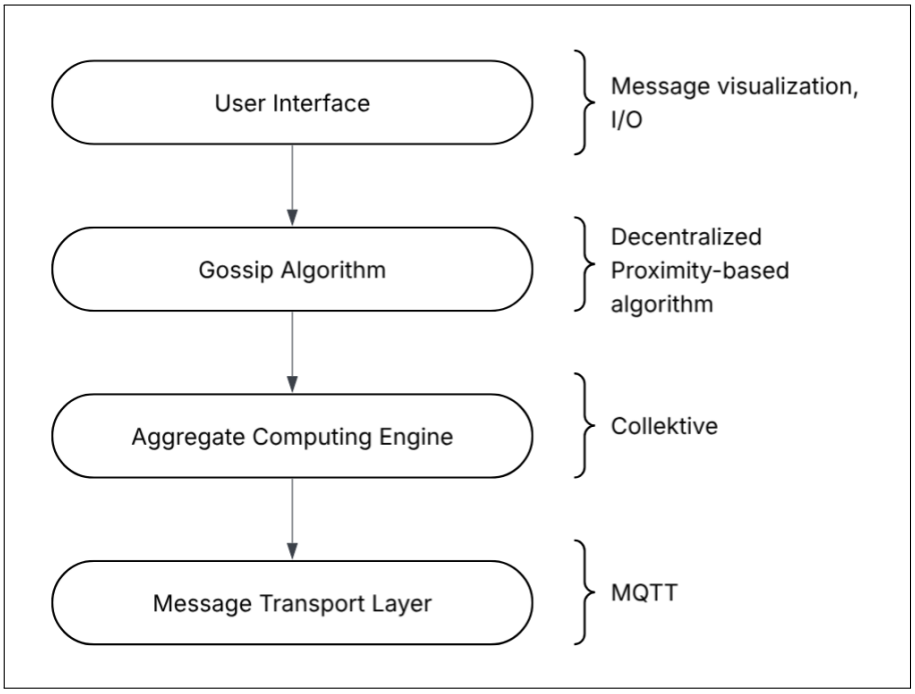

Figure 2.1: High-level overview of the architecture, illustrating the interaction between the different layers of the messaging system.

network represents a single device (e.g., a smartphone) running the messaging software. Each node exhibits the following characteristics:

1. **Lifecycle**: The lifecycle of a node begins when it joins the network and discovers its neighboring devices. While active, the node periodically executes the aggregate algorithm through computation cycles. When the node leaves the network (for example, when the application is closed), it terminates all communication and computation activities.

2. **State**: Each node maintains a local state that includes its unique identifier, a list of neighboring nodes with associated distances, its current message value, and positional information. This state is continuously updated based on local interactions.

3. **Internal Aggregate Computing Logic**: Each node autonomously executes the aggregate algorithm defined in Section 1, which governs how messages are generated, propagated, and expired according to proximity-based

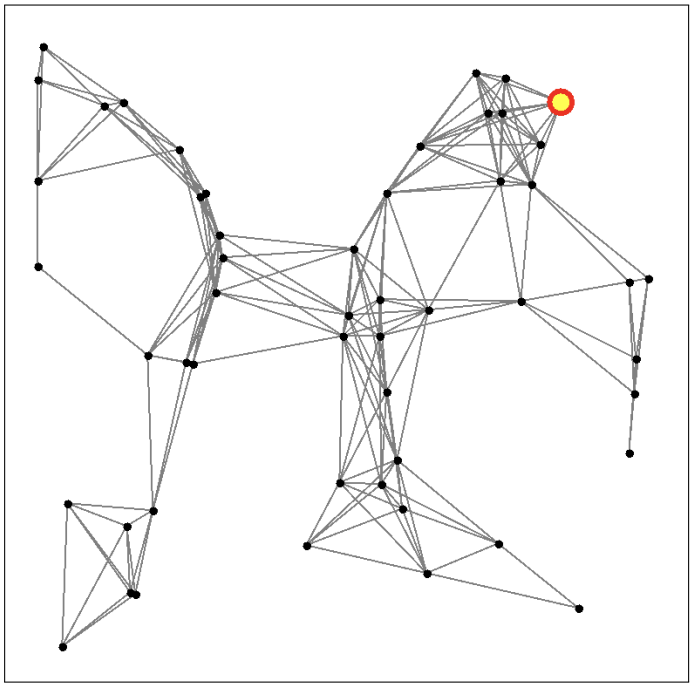

Figure 2.2: Network architecture image taken from *Alchemist Simulator* illustrating nodes and their communication links. Each node represents a device in the messaging system, and edges denote proximity-based connections enabling message exchange.

interactions with neighboring nodes.

The network is **distributed** and **decentralized**, meaning there is no central authority or server managing the communication. All the nodes behave like whole systems, they are considered together as a single entity like a computational field, not as a collection of individual devices. This is achieved through local interactions, each node independently maintains its state and exchanges information with nearby peers, supporting self-organization and resilience.

This type of network is particulary suitable for **dynamic** and **scalable** environments by design, since computation and communication depend only on local interactions and nodes can join or leave the network seamlessly, without any disruption.

2.2.2 Communication Model

TODO: Describe the communication model in detail.

2.3 Technologies

Kotlin Multiplatform

KMP is the programming framework adopted for both the implementation of the messaging algorithm and the development of the mobile application.

Kotlin is a modern, statically typed, cross-platform programming language that supports type inference and emphasizes conciseness, safety, and interoperability. It is fully compatible with Java, as the JVM version of its standard library relies on the Java Class Library, enabling seamless integration with existing Java ecosystems.

The primary goal of Kotlin Multiplatform is to streamline the development of cross-platform applications by allowing developers to share common code across multiple targets (such as Android, iOS, JVM, and Web) while maintaining platform-specific implementations only where necessary. This approach greatly reduces code duplication, accelerates development cycles, and ensures consistency across different environments.

In this project, KMP enables the reuse of the same aggregate computing logic and communication components across both the simulation and mobile application. Compilation targets are defined within the Gradle build configuration, which automates the generation of platform-specific binaries from the shared codebase.

Collektive

Collektive is used as Aggregate Computing framework to implement the messaging algorithm (see Section 1.1.2).

Compose Multiplatform

Compose Multiplatform is the UI framework used to implement the GUI of the application on both Android and iOS. It is based on *Jetpack Compose (JC)*,

Google's declarative toolkit for building native Android interfaces, and extends its capabilities to multiple platforms through the KMP ecosystem.

Compose Multiplatform enables developers to define a single, shared user interface in Kotlin, which is then rendered natively on each target platform. This approach allows the same declarative UI components and logic to be reused across environments, ensuring a consistent design and behavior while maintaining native performance and responsiveness.

In this project, Compose Multiplatform has been adopted to design the messaging application's GUI, allowing users to create, send, and visualize messages within a unified layout. By sharing the same UI code between Android and iOS, the development process is simplified, reducing maintenance overhead and ensuring visual consistency across platforms.

Gradle

Gradle is a powerful build automation and dependency management system widely used in modern software development, particularly for projects based on Java, Kotlin, and Groovy. It is designed for flexibility, scalability, and efficiency, making it one of the most popular tools for managing complex, multi-module applications. Gradle employs a declarative DSL, written in either Groovy or Kotlin, to define build scripts that specify project configurations, dependencies, tasks, and plugins in a concise and human-readable format.

The build process in Gradle is organized around *tasks*, which represent individual units of work, such as compiling source code, running tests, packaging applications, or generating documentation. Developers can define custom tasks and establish dependencies among them, allowing precise control over the execution order and workflow of the build process. Gradle also supports parallel task execution and incremental builds, improving efficiency and reducing build times.

In the context of this project, Gradle is used to configure the Kotlin Multiplatform environment, manage platform-specific dependencies, and generate binaries for multiple targets, including Android, JVM, and iOS. Its integration with the Kotlin DSL ensures a consistent and automated build process across all platforms, simplifying development and maintaining reproducibility across simulation and

deployment environments.

MQTT

MQTT protocol is used as the communication layer to send messages between the various nodes in the distributed network (see Section 1.3.3).

2.4 Messaging with Aggregate Computing

The core logic of the messaging system is implemented using the *Collektive* framework, which provides the necessary abstractions and tools to define aggregate behaviors.

In this section it will be described the algorithm that underpins the system and how it meets the requirements outlined in Section 2.1.1.

2.4.1 Gossip behaviour

Gossip protocols, also known as **epidemic protocols**, are a class of communication algorithms inspired by the way information spreads in social or biological systems. They have emerged as a key communication paradigm for large-scale distributed systems due to their simplicity, scalability, and resilience.

In these protocols, each node periodically contacts one or a few other nodes and exchanges information with them. The dynamics of information propagation closely resemble the spread of an epidemic, leading to high fault tolerance and self-stabilization. The guarantees obtained from gossip-based communication are inherently probabilistic: such systems achieve high stability under stress and disruption, while scaling gracefully to a very large number of nodes [KDG03].

In the context of *Aggregate Computing*, computation focuses on collective behaviours that emerge from local interactions. Each node executes the same local program and exchanges data with its neighbors to collectively produce a global pattern (e.g., diffusion, consensus, or gradient formation). In this sense, *gossip* acts as a mechanism of information diffusion, where nodes iteratively mix their local states with those of their neighbors to approximate global aggregates.

The algorithm developed in this thesis follows a similar **gossip behaviour**, where information spreads through local P2P interactions to collectively achieve a consistent global state in a decentralized and self-stabilizing manner.

Diffusion speed

As defined in [KDG03], the **diffusion speed** of gossip protocols represents the rate at which information originating from any node spreads and becomes uniformly mixed throughout a distributed network. It quantifies how quickly local data diffuses across nodes through randomized peer-to-peer exchanges.

Formally, the diffusion speed $\boldsymbol{T(\delta, n, \epsilon)}$ is defined as the smallest number of rounds required for all (n) nodes' estimates to deviate from the global average by at most a relative error ϵ, with probability at least $1 - \delta$. This metric provides a unified way to reason about the convergence rate of gossip-based communication mechanisms, regardless of the specific network topology.

In the case of **Uniform Gossip**, where each node randomly selects a neighbor to exchange information with in each round, diffusion occurs exponentially fast, with a time complexity of:

$$T(\delta, n, \epsilon) = O(\log(n) + \log(1/\epsilon) + \log(1/\delta)) \tag{2.1}$$

demonstrating that even simple randomized communication mechanisms can achieve rapid and robust global convergence.

2.4.2 Gradient algorithm

The purpose of this algorithm is to enable communication between nodes finding the shortest path to a message source. The algorithm computes, in a fully distributed manner, the minimum distance (**gradient**) from each node to a designated target (the source that sends the message). Full algorithm code is reported in Listing 2.1.

Listing 2.1: Entire algorithm code

```
1 fun Aggregate<Int>.gossipGradient(
2     distances: Field<Int, Double>,
```

```
    target: Int,
    isSource: Boolean,
    currentTime: Double,
    content: String,
    lifeTime: Double,
    maxDistance: Double,
): Message? {

    val isTargetNode = localId == target

    val localContent = if (isTargetNode && isSource) content else ""

    val localDistance = if (isTargetNode) 0.0 else Double.POSITIVE_INFINITY
    val localGossip = GossipGradient(
        distance = localDistance,
        localDistance = localDistance,
        content = localContent,
        path = listOf(localId),
    )

    val distanceMap = distances.toMap()
    val result = share(localGossip) { neighborsGossip: Field<Int, GossipGradient<
        Int>> ->
        var bestGossip = localGossip
        val neighbors = neighborsGossip.toMap().keys

        for ((neighborId, neighborGossip) in neighborsGossip.toMap()) {
            val recentPath = neighborGossip.path.asReversed().drop(1)
            val pathIsValid = recentPath.none { it == localId || it in neighbors }
            val nextGossip = if (pathIsValid) neighborGossip else neighborGossip.
                base(neighborId)
            val totalDistance = nextGossip.distance + distanceMap.getOrDefault(
                neighborId, nextGossip.distance)
            if (totalDistance < bestGossip.distance && neighborGossip.content.
                isNotEmpty()) {
                bestGossip = nextGossip.addHop(totalDistance, localGossip.
                    localDistance, localId)
            }
        }
        bestGossip
    }

    val message = Message(result.content, result.distance)

    return message.takeIf {
        currentTime <= lifeTime &&
            result.distance < maxDistance &&
            result.distance.isFinite() &&
            result.content.isNotEmpty() &&
            localId != target
```

```
48      }
49  }
```

This is a **Bellman-Ford** style algorithm, applied over the network graph:

- **Graph** → undirected weighed graph $G = (V, E)$ where V is the set of nodes and E is the set of edges.
- **Nodes** → each node $u \in V$ represents a device in the network.
- **Weights** → each edge $(u, v) \in E$ has an associated weight $w(u, v)$ representing the distance between nodes u and v.
- **Relaxation** → the algorithm iteratively updates the estimated distances to each node by considering all edges and their weights, gradually converging to the shortest path.

Initialization

The algorithm starts by initializing the nodes. Each node first of all checks if it is the source node (the one that created the message). If it is, it sets its distance to zero and prepares the message content, otherwise it initializes its distance to infinity indicating that it has not yet received the message (Listing 2.2).

Listing 2.2: Initialization code

```
1  val isTargetNode = localId == target
2
3  val localContent = if (isTargetNode && isSource) content else ""
4
5  val localDistance = if (isTargetNode) 0.0 else Double.POSITIVE_INFINITY
6
7  val localGossip = GossipGradient(
8          distance = localDistance,
9          localDistance = localDistance,
10         content = localContent,
11         path = listOf(localId),
12     )
```

Using data class `GossipGradient` (Listing 2.3) to keep track of the current path the message has taken from the origin to the current node with the estimated distance.

Listing 2.3: Data structure representing a state in a gossip-based gradient algorithm.

```
data class GossipGradient<ID : Comparable<ID>>(
    val distance: Double,
    val localDistance: Double,
    val content: String,
    val path: List<ID> = emptyList(),
) {
    /**
     * Reset gossip to start from the local value of the specified node [id].
     */
    fun base(id: ID) = GossipGradient(localDistance, localDistance, content,
        listOf(id))

    /**
     * Add a new hop [id] to the path, update the distance with [newBest] and the
     * localDistance with [localDistance].
     */
    fun addHop(newBest: Double, localDistance: Double, id: ID) = GossipGradient(
        distance = newBest,
        localDistance = localDistance,
        content = content,
        path = path + id,
    )
}
```

Each node starts with `path = listOf(localId)` because the path stored in a node's gossip ends with that node's ID and for every round a neighbor copies your path and then calls `addHop` to add its own ID to the path, so the invariant stays true.

Sharing logic and distance computation

Message dissemination across the network is achieved using Collektive's `share` function, which enables efficient, stateful data sharing among neighboring nodes.

At each computational round k, every node u reads the gossip messages shared by its neighbors (`neighborsGossip`) and updates its local state accordingly. Based on this information, the node applies a relaxation step to compute the minimum distance to the message source. For each neighbor $v \in$ `neighbors`(u), the relaxation is defined as:

$$\boldsymbol{d_u^{k+1} = \min(d_u^k,\ d_v^k + w(v,u))} \tag{2.2}$$

where $w(v, u)$ represents the communication cost or weight of the edge between nodes v and u. This iterative process propagates distance information throughout the network, allowing each node to estimate its proximity to the source node (Listing 2.4).

Listing 2.4: Relaxation code to compute the minimum distance from each node to the source.

```
1 val totalDistance = nextGossip.distance + distanceMap.getOrDefault(neighborId,
      nextGossip.distance)
2 if (totalDistance < bestGossip.distance && neighborGossip.content.isNotEmpty()) {
3     bestGossip = nextGossip.addHop(totalDistance, localGossip.localDistance,
          localId)
4 }
```

This approach is also safe for non-source nodes, which initially have an empty `content` field and an infinite `distance`.

As a result, their state is not propagated to other nodes, since the condition `neighborGossip.content.isNotEmpty()` and the relaxation step prevent the adoption of invalid or uninitialized data. These nodes remain inactive until they receive and relay a valid message proposal, at which point their state becomes part of the propagation process.

Loop avoidance

The `path` field contained in the `GossipGradient` data class is also used to prevent cycles during message propagation (Listing 2.5).

To achieve this, a copy of each neighbor's path is created, excluding the neighbor's own identifier. This produces a sequence representing all the hops that precede that neighbor in the propagation chain.

Two checks are then performed to detect and reject cyclic paths. The first condition, `it == localId`, prevents self-cycles by discarding any proposal that has already passed through the current node. The second, `it in neighbors`, filters out proposals that traverse any of the current node's neighbors other than the one under consideration, thereby avoiding "ping-pong" exchanges between adjacent nodes.

Together, these checks ensure that message propagation remains acyclic, improving the efficiency and correctness of the gossip process.

Listing 2.5: Loop avoidance code preventing cycles during message propagation.

```
val recentPath = neighborGossip.path.asReversed().drop(1)

val pathIsValid = recentPath.none { it == localId || it in neighbors }

val nextGossip = if (pathIsValid) neighborGossip else neighborGossip.base(
    neighborId)
```

Message expiration and spatial limits

To regulate both the lifespan and spatial reach of messages within the network, the algorithm takes as input two key parameters: the Time To Live (TTL) (`lifeTime`) and the maximum propagation distance (`maxDistance`). Before forwarding a message, each node applies a filtering step to ensure that only valid messages, those that have not expired and remain within the defined spatial range, are relayed to neighboring nodes (Listing 2.6).

Listing 2.6: Filtering logic enforcing message expiration and spatial propagation limits.

```
return message.takeIf {
        currentTime <= lifeTime &&
            result.distance < maxDistance &&
            result.distance.isFinite() &&
            result.content.isNotEmpty() &&
            localId != target
    }
```

Multiple sources

In a large-scale, real-world deployment, it is common for multiple nodes to generate and transmit messages simultaneously. The algorithm is designed to handle this scenario effectively, allowing multiple sources to coexist and propagate their messages independently without interference.

This behavior is achieved by treating each message source as an independent entity, with every node maintaining separate state information for each active message in the network. Initially, a gossip-based self-stabilization step is applied to the set of active sources (Listing 2.7).

Listing 2.7: Gossip-based self-stabilization of the message sources.

```
val localSources: Set<Int> = if (isSource) setOf(localId) else emptySet()
val sources: Set<Int> = share(localSources) { neighborSets: Field<Int, Set<Int>>
    ->
    neighborSets.neighborsValues.fold(localSources) { accumulator, neighborSet ->
        accumulator + neighborSet
    }
}
```

After stabilization, the gossip-gradient algorithm described previously is executed independently for each source (Listing 2.8).

Since the implementation relies on the `share` construct, iterating over all sources requires the use of the `alignedOn` operator. This operator ensures that only nodes executing the same `alignedOn(sourceId)` instance exchange values with one another, thereby preventing interference between computations associated with different sources.

Listing 2.8: Execution of the gossip-gradient algorithm for multiple concurrent sources.

```
val messages = mutableMapOf<Int, Message>()
    for (sourceId in sources) {
        alignedOn(sourceId) {
            val result = gossipGradient(
                distances = distances,
                target = sourceId,
                isSource = localId == sourceId && isSource,
                currentTime = currentTime,
                content = content,
                lifeTime = lifeTime,
                maxDistance = maxDistance,
            )

            result?.let { messages[sourceId] = result }
        }
    }
```

Time complexity

Let $n = |V|$ be the number of nodes, $m = |E|$ the number of links, and s the number of sources. The gradient phase is Bellman-Ford-like with non-negative weights, the pure distance relaxation costs $O(nm)$ per source in the worst case (thus $O(n^3)$ on dense graphs with $m \approx n^2$).

In the implementation presented here, the *loop-avoidance* check scans the neighbor's path prefix, adding an $O(n)$ factor per neighbor. This yields $O(mn)$ work per round and up to $O(n)$ rounds, i.e., $O(mn^2)$ per source (dense: $O(n^4)$).

Across s sources, multiply the bounds: $O(snm)$ (dense: $O(sn^3)$) for the pure gradient, or $O(smn^2)$ (dense: $O(sn^4)$) with full path checking. Bounding the path inspection to a constant k (or removing it) recovers the $O(snm)$ worst-case bound.

2.5 Mobile Application

The mobile application serves as the GUI of the messaging system, providing users with both an intuitive interface and a runtime environment for executing the algorithm.

Echo is the name chosen for the application, as it evokes the idea of messages reverberating through the network within a limited range and time frame, analogous to how sound echoes in space. The name also recalls the Linux command `echo`, used to print messages to the terminal, symbolizing communication and simplicity.

The application is implemented using KMP and Compose Multiplatform, ensuring **cross-platform** compatibility between Android and iOS devices. This section describes the architecture and implementation details of the app.

2.5.1 Architecture

The architecture of the **Echo** application is designed to be modular and scalable, enabling easy maintenance and future extensions. It follows the **Model-View-ViewModel (MVVM)** architectural pattern, which separates the user interface, business logic, and data management into distinct layers (Figure 2.3).

The MVVM pattern consists of three main components:

- **Model**: Represents the data and business logic of the application. It is responsible for managing data sources, such as local databases or APIs, and performing necessary transformations or computations.

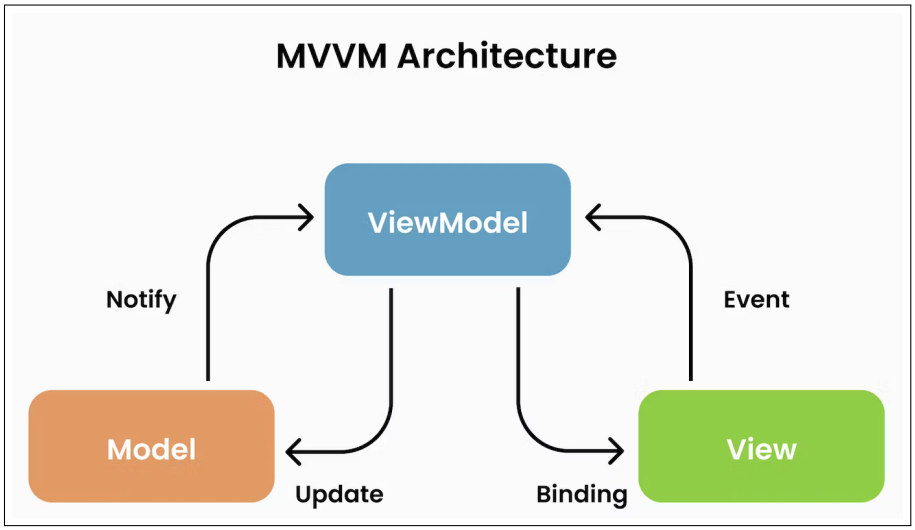

Figure 2.3: Illustration of the MVVM architectural pattern.

- **View**: Defines the UI layer—the part of the application that the user interacts with. It displays data exposed by the ViewModel and forwards user interactions (e.g., button clicks or text input) back to it.
- **ViewModel**: Acts as an intermediary between the Model and the View. It retrieves and processes data from the Model, exposing it in a UI-friendly form. The ViewModel maintains the application state and employs observable data structures to notify the View of any updates.

2.5.2 Layered design

The layered design of the application is illustrated in Figure 2.5. Each layer is described below following a bottom-up approach.

Collektive Engine. This layer acts as the Aggregate Computing runtime environment responsible for managing and executing aggregate programs, including the messaging algorithm. It provides primitives for defining computational fields, sharing data among neighbors, and performing distributed computations across the network, such as `share`, `alignedOn`, and `neighboring` (Listing 2.9).

Listing 2.9: Collektive engine setup.

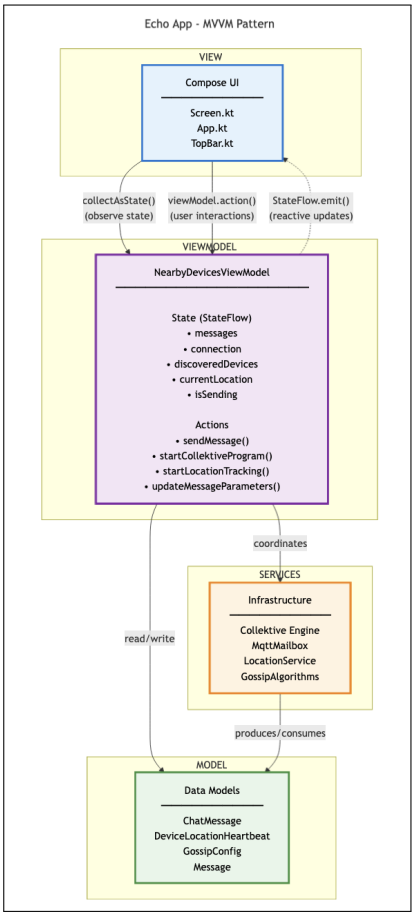

Figure 2.4: MVVM architecture of the **Echo** mobile application.

```
Collektive(
    deviceId,
    MqttMailbox(...),
) {
    // aggregate program runs on top of this engine
    val neighborMap = neighboring(localId)
    val result = share(...) { ... }
    val allSourceMessages = chatMultipleSources(...)
}
```

The **Location Services** layer provides access to the device's native location APIs to obtain accurate positional information (Listing 2.10). It captures latitude and longitude values, essential for computing distances between nodes in the messaging algorithm, and updates the device's location in real time.

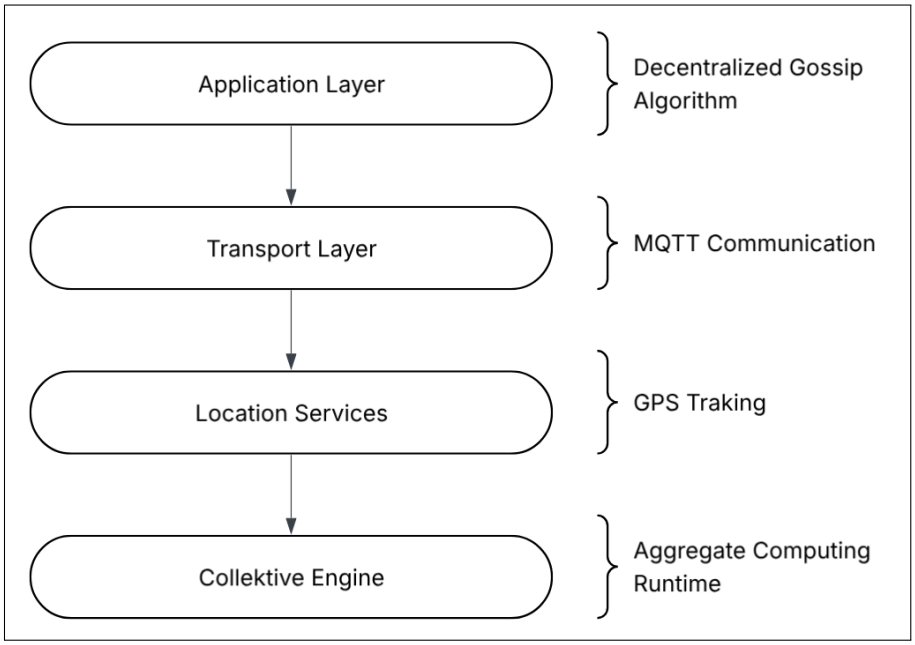

Figure 2.5: Layered design of the mobile application, illustrating the interaction between the different components.

Listing 2.10: Data class for GPS location.

```
@Serializable
data class DeviceLocation(
    val latitude: Double,
    val longitude: Double,
    val accuracy: Float?,
    val timestamp: Long
)
```

To compute distances between two devices with known coordinates, the Haversine formula is used:

$$d = 2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\Delta\lambda}{2}\right)}\right) \tag{2.3}$$

The location data obtained locally is shared through the **Transport Layer**, enabling neighbor awareness without direct coupling to the Collektive Engine. This layer uses the MQTT protocol as its communication mechanism, relying on the public broker *test.mosquitto.org* over WebSocket on port 8080 instead of the default TCP port 1883. Using port 8080 improves firewall compatibility, since

many private or public networks block non-standard ports like 1883, while 8080 is commonly allowed for web traffic. This setup also ensures compatibility with browsers and WebApps, facilitating future extensions of the application to web platforms.

The Transport Layer is responsible for the following functionalities:

- **Neighbor discovery**: Each device periodically broadcasts its presence via a heartbeat message, indicating active network connectivity, to the wildcard topic `echo/heartbeat/+` (Listing 2.11).

Listing 2.11: Heartbeat pulse recursive sending function.

```
private suspend fun sendHeartbeatPulse() {
    ...
    mqttClient?.publish(
        retain = false,
        qos = Qos.AT_MOST_ONCE,
        topic = heartbeatTopic(deviceId),
        payload = payload,
    )
    delay(1.seconds)
    sendHeartbeatPulse()
}
```

The QoS level 0 (*fire and forget*) is used, since heartbeats are ephemeral and frequent, minimizing overhead. A timeout mechanism is implemented to remove inactive neighbors (Listing 2.12): if a device stops sending heartbeats (due to crash, disconnection, or moving away) for more than 5 seconds (`retentionTime`), it is removed from the neighbor list.

Listing 2.12: Heartbeat pulse cleanup function.

```
private suspend fun cleanHeartbeatPulse() {
    cleanupNeighbors(retentionTime)
    delay(retentionTime)
    cleanHeartbeatPulse()
}
```

- **Message delivery**: Each device subscribes to its dedicated topic `echo/messages/<deviceId>` to receive messages. For this purpose, QoS level 1 (*at least once*) is used to ensure reliable delivery:

1. The sender publishes a message to the broker.
2. The broker acknowledges receipt (`PUBACK`).
3. The broker forwards the message to subscribers.
4. The subscriber acknowledges receipt (`PUBACK`).
5. If acknowledgment fails, the message is retransmitted.

- **Location sharing**: GPS coordinates are embedded within the heartbeat payloads (Listing 2.13), allowing nodes to discover neighbors and compute distances without requiring a separate location-sharing mechanism.

Listing 2.13: GPS coordinates embedded inside the heartbeat payload.

```
@Serializable
data class DeviceLocationHeartbeat(
    val deviceId: String,
    val location: DeviceLocation?,
    val timestamp: Long,
)
```

This design ensures that GPS data is continuously updated through the heartbeat frequency.

Finally, the **Application Layer** encapsulates the core logic of the messaging system. It implements the decentralized gossip algorithm responsible for message propagation and handles all user-facing features, including message creation, parameter configuration (expiration time and propagation distance), and real-time visualization of messages.

Chapter 3

Validation

This chapter describes the validation process undertaken to ensure the correctness of the algorithm. It's divided into Continuous Integration (CI) setup and Simulation experiments.

3.1 Continuous Integration

CI is a software development practice that involves automatically building, testing, and validating code changes as they are integrated into a shared repository. The integrations are made frequently, ensuring that the system remains in a consistently deployable state throughout the development lifecycle. This approach helps maintain code quality, reduces integration problems, and promotes collaboration among team members.

In this project, **GitHub Actions** is used as the CI platform to automate the build processes for both the shared codebase [1] and the mobile application. The CI pipeline is configured to trigger on every push or pull request to the main branch, ensuring that all changes are automatically validated before being merged.

The algorithm was added to Footnote 1, repository which hosts various examples of applications built using the Collektive framework. The CI workflow includes the following jobs (Listing 3.1):

1. `build`: it's the main job of the workflow, responsible for the compilation,

[1] `https://github.com/Collektive/collektive-examples`

Listing 3.1: CI workflow for Collektive examples.

```yaml
jobs:
  build:
    strategy:
      matrix:
        os: [windows, macos, ubuntu]
    runs-on: ${{ matrix.os }}-latest
    concurrency:
      group: build-${{ github.workflow }}-${{ matrix.os }}-${{ github.event.number
          || github.ref }}
      cancel-in-progress: true
    steps:
      - name: Checkout
        uses: actions/checkout@v5.0.0
      - name: Check the simulations on CI
        uses: DanySK/build-check-deploy-gradle-action@4.0.10
        with:
          build-command: true
          check-command: ./gradlew check runAllGraphic
          deploy-command: true
          should-run-codecov: false
          should-validate-wrapper: ${{ contains('Linux', runner.os) }}
  success:
    runs-on: ubuntu-latest
    needs:
      - build
    if: >-
      always() && (
        contains(join(needs.*.result, ','), 'failure')
        || !contains(join(needs.*.result, ','), 'cancelled')
      )
    steps:
      - name: Verify that there were no failures
        run: ${{ !contains(join(needs.*.result, ','), 'failure') }}
```

testing and deployment over three operating systems—Windows, Ubuntu, and macOS. Each run begins by checking out the project's source code from the repository, after which the workflow executes the Gradle build and testing phases through the `build-check-deploy-gradle-action`. This action automates the project verification process by running commands such as `./gradlew check runAllGraphic`, ensuring that the simulations and validation steps complete successfully.

2. `success`: executed after the `build` job completes, serves as a verification step, it checks whether all builds have finished successfully or if any have failed or been cancelled.

3.2 Alchemist simulations

Deploing and testing the messaging system in a real-world environment with many mobile devices can be logistically challenging and require significant resources. To overcome these challenges, the *Alchemist Simulator* is used to create a controlled environment with multiple nodes where various scenarios can be simulated and analyzed.

The simulations executed through the `Alchemist Simulator` are configured using **YAML** files, which specify the parameters governing the simulation environment, the number and type of nodes, and the associated behavioral models. The example shown in Listing 3.2 presents a representative configuration defining a simulation scenario with 50 regular nodes and 2 source nodes.

Each node executes the program `gossipChatMultipleSourcesEntrypoint` (Listing 3.3), which serves as the entry point of the simulation. This component initializes the *gossip-based chat algorithm*, invoking it with predefined parameters such as the message lifetime and the maximum propagation distance, thereby reproducing the intended proximity-based communication dynamics.

Using the GUI provided by the Simulator, it is possible to visualize the network topology and change it, monitor the state of each node in real time and analyze how messages propagate through the network over time.

Listing 3.2: YAML configuration file for the Alchemist simulation.

```yaml
network-model:
  type: ConnectWithinDistance
  parameters: [5]

incarnation: collektive

_pool: &program
  - time-distribution: 1
    type: Event
    actions:
      - type: RunCollektiveProgram
        parameters: [it.unibo.collektive.examples.chat.GossipChatMultipleSourcesKt
            .gossipChatMultipleSourcesEntrypoint]

deployments:
  - type: Rectangle
    parameters: [50, 0, 0, 20, 20]
    programs:
      - *program
    contents:
      - molecule: source
        concentration: false
  - type: Point
    parameters: [10, 10]
    programs:
      - *program
    contents:
      - molecule: source
        concentration: true
  - type: Point
    parameters: [ 15, 15 ]
    programs:
      - *program
    contents:
      - molecule: source
        concentration: true
```

Listing 3.3: Entry point for the multi-source chat simulation.

```kotlin
fun Aggregate<Int>.gossipChatMultipleSourcesEntrypoint(
    environment: EnvironmentVariables,
    distanceSensor: CollektiveDevice<*>,
): String {
    val distances: Field<Int, Double> = with(distanceSensor) { distances() }
    val isSource = environment.get<Boolean>("source")
    val currentTime = distanceSensor.currentTime.toDouble()

    return chatMultipleSources(
        distances,
        isSource,
        currentTime,
    ).map { "${it.key}: ${it.value.content}" }
        .joinToString("\n")
}
```

3.2.1 Simulation with GPS traces

`Alchemist` is equipped with the ability to load and simulate on real-world maps. This feature allows for more realistic simulations by incorporating actual geographic data and node placements.

The data of Vienna's 2021 marathon event was used to create a realistic challenging environment to test the messaging system Figure 3.1.

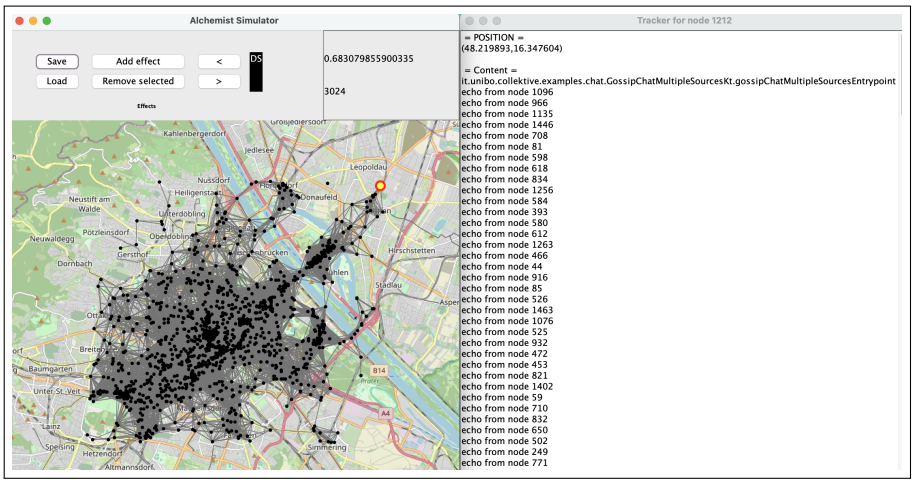

Figure 3.1: Simulation of the messaging system using real-world GPS traces from Vienna's 2021 marathon event.

Chapter 4

Conclusions

4.1 Final considerations

4.2 Future work

Bibliography

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015.

[Cor24] Angela Cortecchia. *A Kotlin multi-platform implementation of aggregate computing based on XC*. PhD thesis, University of Bologna, 2024.

[Cor25] Angela Cortecchia. Collektive documentation, 2025.

[CVAP22] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafi: A scala dsl and toolkit for aggregate programming. *SoftwareX*, 20:101248, 2022.

[FPBV17] Matteo Francia, Danilo Pianini, Jacob Beal, and Mirko Viroli. Towards a foundational api for resilient distributed systems design. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 27–32, 2017.

[Inc14] Bridgefy Inc. Bridgefy: Offline messaging app, 2014.

[KDG03] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 482–491, 2003.

[MTN+17] Luke P. Morrison, Brian Team, Brian Nguyen, Senthil Kannan, Nathan Ray, and Gregory C. Lewin. Airchat: Ad hoc network monitoring with drones. In *2017 Systems and Information Engineering Design Symposium (SIEDS)*, pages 38–43, 2017.

[PMV13] D Pianini, S Montagna, and M Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, August 2013.

[PVB15] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, page 1846–1853, New York, NY, USA, 2015. Association for Computing Machinery.

[RSG+11] Michael Rogers, Eleanor Saitta, Torsten Grote, Julian Dehm, Mikolai Gütschow, Nico Alt, and Briar Project. Briar: Secure messaging anywhere, 2011.

[SBEK16] Massinissa Saoudi, Ahcène Bounceur, Reinhardt Euler, and Tahar Kechadi. Energy-efficient data mining techniques for emergency detection in wireless sensor networks. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6, 07 2016.

[Usm21] Mohammad Faiz Usmani. Mqtt protocol for the iot - review paper, 05 2021.

[VBD+19] Mirko Viroli, Jacob Beal, Francesco Damiani, Stefano Montagna, Danilo Pianini, Luca Ricci, and Franco Zambonelli. *Aggregate programming: from foundations to applications*. Springer, 2019.