Communication in D-MANETs using Tuple Spaces

By: Juan Camilo Bages   Promoter: Nicolas Cardozo

December 2018

Abstract

Mobile devices can be used for creating networks in which hosts can communicate directly between each other in an infrastructure-less environment. However, important challenges arise when working with these networks. In this thesis, we propose a framework for abstracting the development of applications that communicate directly. This framework takes advantage of the simplicity of the Tuple Space model in order to hide complex communication logic from the application layer.

Contents

Chapter 1

Introduction

1.1 Context

Since the rise of the mobile device era, almost every person in the world is the owner of a smartphone. These little yet powerful devices now come equipped with multiple core processors, gigabytes of primary and secondary storage, and a variety of communication mechanisms including Wi-Fi, Bluetooth, and mobile radio signal. Apart from that, they come with a rich set of APIs for developing any kind of application using popular programming languages such as Java, Kotlin, and Swift.

As these devices are portable, they can be used for creating networks in which hosts can communicate directly between each other without the need of some centralized authority or infrastructure. However, important challenges arise when working with this kind of networks, as devices can connect and disconnect with each other in unpredictable ways [1]. In addition to that, these devices are still limited in resources compared to desktops or laptops, and they impose new restrictions such as battery life, which can be crucial for the device's owner.

1.2 Problem Statement

Infrastructure-based networks provide an effective mechanism for connecting mobile devices. Protocols such as TCP rely on the fact that there is a strong supportive infrastructure that guarantees certain quality assumptions. Some of these assumptions are short delay times and a permanent connection between the communicating parts [2]. However, setting up an infrastructure is expensive, time consuming, and not always possible. Two well-known alternatives for this are Mobile ad-hoc networks (MANETs) and Delay Tolerant Networks (DTNs) [1]. These networks propose methodologies in which mobile devices communicate directly following some strategy or protocol. In this thesis we explore these networks and how combining them with the Tuple Spaces model can lead us to an easier way to develop applications with infrastructure-less communication capabilities.

1.3 Approach

In this thesis we create a reusable framework for developing applications that communicate using Tuple Spaces. This framework allows applications to communicate in an infrastructure-less way as they exchange messages directly, as MANETs and DTNs propose. Our framework will use an epidemic strategy for disseminating messages and will provide an abstraction below the application layer for the developer in a way that she can focus on the application logic. In addition to this we will show some examples of applications we developed using this framework and show how it makes communication mechanisms transparent for the developer.

1.4 Contributions

This thesis makes the following contributions to the area of distributed computing:

- A reusable framework for developing applications that communicate over infrastructure-less networks using Tuple Spaces.

- A demonstration of the framework usage on the implementation of two example applications that communicate over an infrastructure-less network using Tuple Spaces.

1.5 Outline

The remainder of this thesis goes as follows. On Chapter 2 we will give some background about Tuple Spaces, infrastructure-less networks, routing protocols for these networks, and communication mechanisms for exchanging information between devices directly. Then, on Chapter 3, we will outline the architecture of the framework we developed and explain the most relevant implementation details. Later, on Chapter 4, we will talk about the example apps we created, show how the framework was useful in order to develop them, and evaluate how these applications behaved in a test environment. Finally, on Chapter 5, we will show conclusions of the thesis.

Chapter 2

Background

In this chapter we discuss some concepts required for understanding this thesis. In the first section we present Tuple Spaces, their origin, and how they provide a simple yet powerful model for concurrency and parallelism. In the second section, we present Mobile Ad-Hoc Networks (MANETs) along with their advantages and disadvantages. In the third section, we present Delay Tolerant Networks (DTNs) along with their advantages and disadvantages. In the forth section, we present Disconnected Mobile Ad-Hoc Networks (DMANETs). Finally, in the fifth section, we present the epidemic routing protocol.

2.1 Tuple Spaces

2.1.1 Summary

Tuple Spaces (TS) were first introduced as part of the Generative Communication model for the Linda programming language presented by David Gelernter in 1985. He refers to TS as "The abstract computation environment that is the basic of Linda's model of communication" [3]. Programs in Linda communicate by sharing a single TS in which they can add, remove, and read tuples in an atomic way. For example, consider two processes A and B trying to communicate. In order for A to send data to B, it must generate tuples with the data and insert them into the TS. In the same way, process B will try to withdraw data in the form of a tuple directly from the TS [3]. With that said, TS acts as a centralized atomic data structure in which processes can exchange information without knowing about each other and without requiring to be available at the same time. This means that TS provide a model for concurrency that is decoupled both in time and space [4].

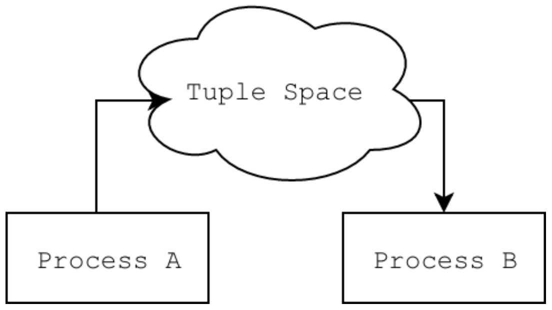


Figure 2.1: Process A communicating with process B using a Tuple Space.

2.1.2 Tuples

A tuple or n-tuple is defined as an ordered sequence of (possibly duplicate) elements. Tuples can be arbitrarily long and can hold any type of element [5]. Some examples of tuples are the following:

$()$	Empty tuple or 0-tuple
$(1, 2, 64, 12)$	Tuple containing only integers
$(''Hey'', 434, true)$	Tuple containing string, integer, and boolean
$(? : int, ? : int)$	Tuple containing only integer formals

In TS, each element of the tuple can be of one of two types: formals or actuals [3]. Actuals refer to elements that are defined such as the ones shown in the example above (except for the last tuple). Formals, refer to ”wild-cards” or placeholders, like the ones we see in the last example above. We will discuss how these formals and actuals types interact with each other when we talk about TS matching.

2.1.3 Interface

TS provide a simple yet powerful interface for doing tuple operations. One of the guarantees that TS provide are the atomicity of each of its operations. The operations we can perform over a TS are the following:

in(tuple)

The in operation receives a tuple and tries to retrieve some tuple from the TS matching the one given as an argument [3]. The details about matching are discussed below. In case there is a match, the tuple will be removed from the TS and returned to the calling process. On the other hand, in case there is no matching tuple available in the TS, the calling process will suspend until some other process inserts a matching tuple into the TS. In case there are multiple options that match the given tuple, any of them could be returned [6]. Additionally, if multiple processes try to acquire the same matching tuple, only one of them will receive it while the others will suspend and wait for a new matching tuple to be inserted in the TS [3].

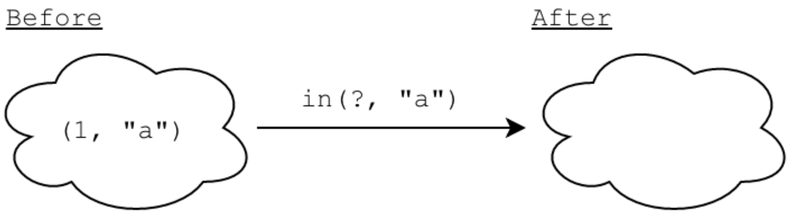


Figure 2.2: Tuple Space before and after calling in (...) .

read(tuple)

The read operation is very similar to the in operation with the difference that in the case of a match, read does not remove the tuple from the TS [3]. As the read operation does not modify the TS, multiple processes can access it without blocking each other. However, in the case that no matching tuple is found in the TS, the calling process will suspend in the same way as the in operator.

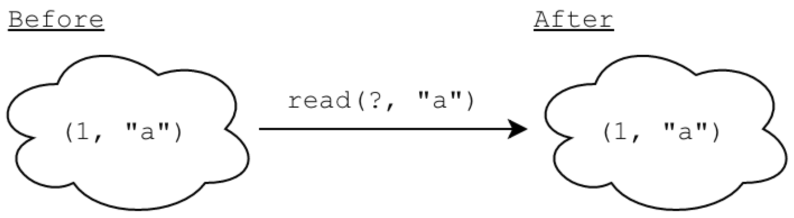


Figure 2.3: Tuple Space before and after calling read (...) .

out(tuple)

The out operation receives a tuple and insert it into the TS [3]. Multiple tuples having the same form can exist at the same time in the TS so no validation is required. When this operation is performed, suspended processes waiting for a tuple in the TS that match the one that was just inserted will be awaken and try to remove or read it depending of the operation they called [3].

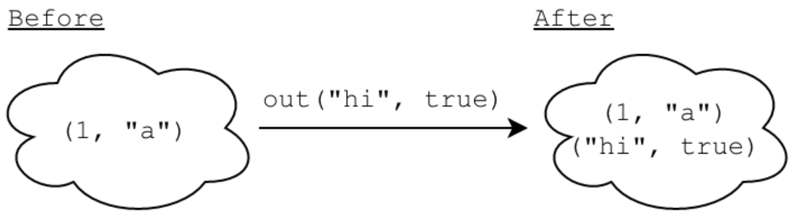


Figure 2.4: Tuple Space before and after calling out (...) .

2.1.4 Matching

Two tuples A and B will match and return a tuple C if and only if the following conditions are true. First, both tuples are required to have the same number of elements. Otherwise, some elements from the longer tuple would not match with the shorter tuple. Second, each element of tuple A must match each element of tuple B in an ordered manner. This condition is quite complex as elements in the tuple can be formals or actuals, so the following scenarios can happen:

- Actual with actual: Two tuples match if and only if both elements have the same type and value [3]. For example, element 2 will match with element 2 but not with element "2".
- Actual with formal: Two tuples match if and only if the formal is of the same type as the actual [3]. For example, a formal of type integer will match the element 2 but not the element "2". When this scenario happen, the tuple C resulting from the match will have the actual element in this position.
- Formal with formal: Two tuples composed of formals only will never match. Formals cannot match other formals [3].

In the following table we can see some examples of tuple matching with their respective result:

Tuple 1	Tuple 2	Result
("x", 12)	("x", 12)	("x", 12)
("x", ?: int)	("x", ?: int)	No match
(?: string, 12)	("x", ?: int)	("x", 12)
("x", 12)	("x", 12, ?: int)	No Match
("x", 12)	("y", 12)	No Match

Table 2.1: Examples of tuples trying to be matched and their result.

2.2 Mobile Ad-Hoc Networks

Mobile Ad-Hoc Networks (MANETs) are dynamic wireless networks with little or no fixed infrastructure [1]. These networks are different from traditional infrastructure-based networks as nodes belonging to it communicate directly [4]. MANETs provide a cost efficient way to communicate mobile devices and a powerful alternative for scenarios that will not allow infrastructure, for example, disaster recovery communication. As nodes may move freely, the network topology may change rapidly and unpredictable.

MANETs routing protocols can be classified into proactive and reactive protocols. Proactive protocols actively propagate route updates in order to maintain updated information of the topology across the network. On the other hand, reactive protocols try to establish a route from source to destination only when needed [1]. The disadvantage with MANETs routing protocols for this thesis is that they assume that there exist a connected path from source to destination. This is a problem because there are environments in which connected paths may never exist between some pairs of nodes [4].

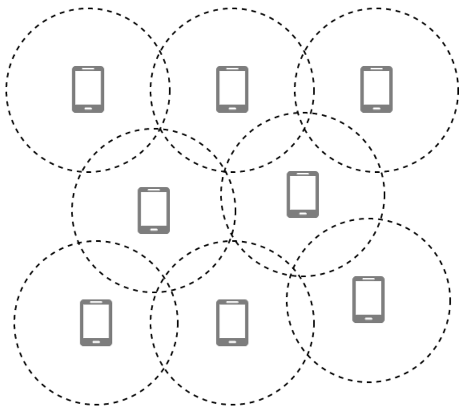


Figure 2.5: MANET showing the communication between 8 devices.

2.3 Delay Tolerant Networks

Delay Tolerant Networks (DTNs) are networks that attempt to provide communication in environments characterized by long delay paths and frequent network partitions. These type of environments are known as "challenged networks" and are not suitable for traditional Internet protocols. Because of the nature of these environments, they may never have a connected path between some pairs of nodes [1]. Even though DTNs take into account this type of environments in which MANETs protocols may fail, some approaches make the assumption that communication between nodes can be predicted accurately. However, this can be a problem as there are scenarios in which nodes can move in a randomly or difficult to predict way [4]. One case in which this happen is when nodes are laptops or mobile phones being carried by humans.

DTNs are intended to operate as a store-and-forward gateway on top of different protocols. This way DTNs introduce the concept of a region and a gateway. The purpose of gateways or DTN gateways is to communicate different regions trying to talk between each other. Regions are parts of the network where no DTN gateway is required in order to communicate between pairs of nodes. With that said, each individual region can have some communication stack that may be different from that of other regions. DTN gateways make regions communication feasible [1].

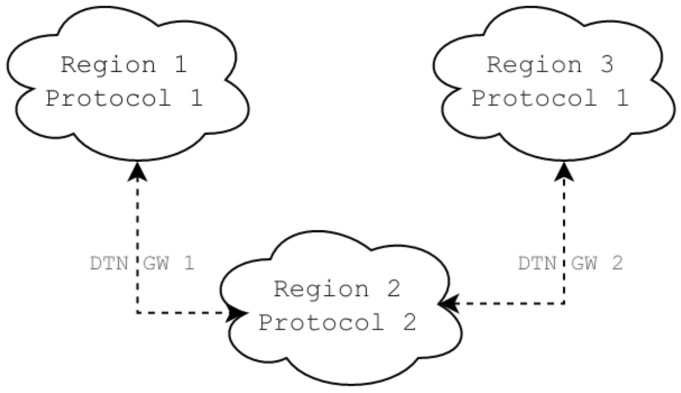


Figure 2.6: DTN with 3 regions connected with 2 DTN gateways.

2.4 Disconnected MANETs

Disconnected MANETs (DMANETs) are networks with little or no fixed infrastructure. This type of networks contain a sparse population of nodes, and as a result it is divided into a set of disjoint components or islands [4]. Because of this, it may be impossible to have a connected path between some pairs of nodes. However, because of the dynamic nature of these networks, a non-direct path may be formed over the time. This is something routing protocols can take advantage from as data can propagate from node to node in a time decoupled manner until they get to its destination [1].

The dynamic nature of DMANETs propose new challenges as there is no node in the network that can be considered stable. This is because pairs of nodes may connect and disconnect in unpredictable ways, and there is no fixed infrastructure to support the communications in contrast with infrastructure-based networks. Furthermore, synchronous communication becomes unfeasible as delays in propagation can be arbitrarily long and it can be the case that data never arrive [4].

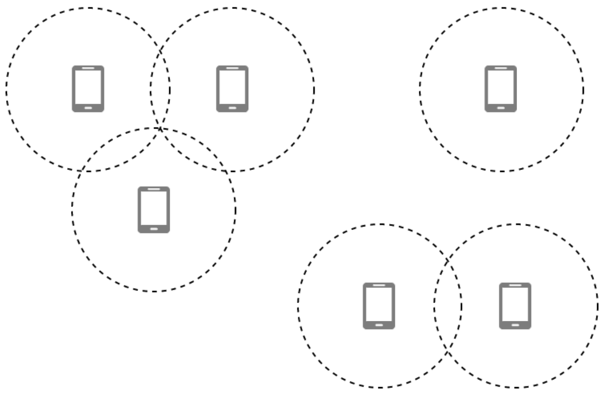


Figure 2.7: DMANET with 6 devices in 3 islands.

2.5 Epidemic Routing Protocol

2.5.1 Summary

Epidemic Routing is a protocol for communicating nodes in partially connected ad-hoc networks. The basic idea consist of exchanging messages between random-wise pairs of nodes in order to maximize the number of delivered messages [7]. Take as an example a network consisting of 3 nodes A, B, and C. Node A is trying to send a message to node C but there is not a direct connection between them. However, node A is connected with node B so it can send the message to it first. Then, at a later time, node B will meet node C and deliver the message to it.

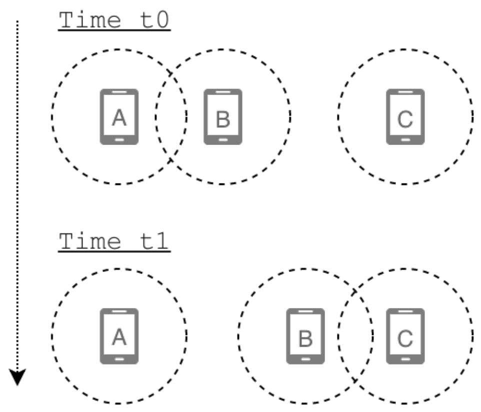


Figure 2.8: Indirect path between nodes A and C using node B.

2.5.2 Protocol

The epidemic routing protocol works as follows. Each host has an identifier and maintains a buffer of messages it has generated as well as messages it is storing from other nodes. Each host also stores a bit vector called the summary vector that indicates which messages they have saved locally [7]. Using this vector messages are not trapped in a loop between two nodes as they are not requested if they were already stored. When two nodes get close enough for communicating, the node with the smaller identifier starts a communication session in order to exchange the messages each node is missing and the other node have. As an optimization, each node can have a cache of nodes it recently communicated with in order to avoid exchanging messages again until some time threshold is exceeded [7]. Listing 2.1 shows a simplified pseudo-code of the protocol.

2.5.3 Communication Session

During the first step of the communication session, each node exchange its summary vector with the other node. Then, as a second step, each node compare the summary vector it received with its own summary vector to determine which messages it is missing and it can retrieve from the other node. This step can be done using the AND operation between the first summary vector and the compliment of the second summary vector. Finally,

```
id = get_host_id()

recent_devices = []
summary_vector = []

fun epidemic_routing(nearby_host) do
    # Start session if I has the smaller id
    if id > nearby_host.id do
        return
    end

    # Avoid communicating with recent devices
    if nearby_host.id in recent_devices do
        return
    end

    # Add to recent devices
    recent_devices.add(nearby_host)

    # Start a communication session
    start_communication_session()
end
```

Listing 2.1: Protocol pseudo-code

as a third step, each node request the other node to send the messages it is willing to receive and the communication session ends when this message exchange is finished [7].

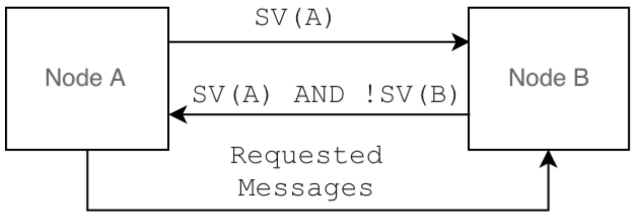


Figure 2.9: Communication session between 2 nodes A and B.

2.5.4 Messages

Each message in the protocol has associated a unique message identifier, a hop count, and an optional ACK request. The message identifier is defined as a unique 32-bit integer with the first half being a 16-bit node identifier and the second half being a locally generated message ID [7]. The hop count is a value similar to the TTL and it defined how many hops the message can have before reaching its destination. For example, a hop count of one means that the next hop of the message needs to be to its destination. Furthermore, every time the message jumps from some node A to some other node B, the hop count value is decreased by one [7].

Chapter 3

Framework Architecture

In this chapter we discuss about both the framework architecture and give some implementation details. In the first section we will give a basic overview on the layer architecture of the framework. Finally, in the second, third, and forth sections, we will dive into the details of each layer we implemented.

3.1 Overview

The framework is organized in a layered architecture with 3 layers. These layers are the Tuple Space layer, the routing layer, and the communication layer. Each layer provides an interface for communication and will interact only with adjacent layers. The framework is intended to operate below the application layer and the developer's application will only interact with the top-most layer of the framework, that is, the Tuple Space layer. Figure 3.1 shows the organization of the layers in the framework. In order to provide implementation flexibility, each layer will expose an interface in such a way that specific implementations can be specified without major changes to the code. It is worth to mention that the language we chose for implementing the framework was The Java Programming Language because it is compatible with multiple platforms including PCs and Android.

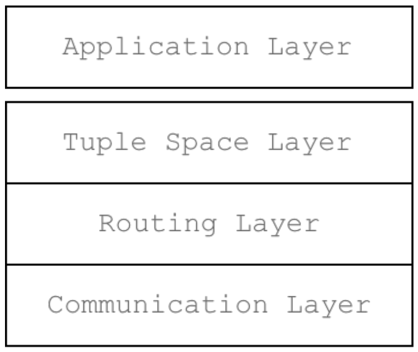


Figure 3.1: Framework architecture layers.

3.2 Tuple Space Layer

The Tuple Space layer is the top-most layer of the stack. It provides the same operations as the TS interface we described in the previous chapter. In the following subsections we explore the parts that compose this layer.

3.2.1 Tuple Space Interface

This is the main part of the Tuple Space layer. It is the entry point for interaction from both the application layer and the routing layer. From the developer's perspective, all the operations are done locally with the Tuple Space without knowledge. Listing 3.1 shows the interface we provide for the developer in order to interact with the framework.

As we mentioned before, DMANETs will mostly have large delays and some messages may never arrive. With that said, the traditional approach of TS in which processes sleep in a synchronous way until a matching tuple is inserted will not work in this case. In order to handle this, asynchronous communication is introduced for both the in and read operations using Future constructs. Futures provide a placeholder for a value that is still not available [4]. In this case, the object that is still not available is the tuple requested by the calling process in both the in and read operations. On the other hand, the out operation will only write to its local TS so this operation can be performed synchronously just by writing into the tuples container. As an

```java
public interface ITupleSpace {

    void out(ITuple tuple);

    void outMany(ITuple... tuple);

    void outRouting(ITuple... tuples);

    Future<ITuple> in(ITuple tuple);

    Future<ITuple> read(ITuple tuple);

}
```

Listing 3.1: ITupleSpace interface

extra, a variation of the out method was added for adding multiple tuples at once from the application layer and from the routing layer.

3.2.2 Tuple Interface

A tuple represents the basic object that the Tuple Space will hold and devices will use for communicating and exchange information. Developers will be able to interact with tuples by inserting or retrieving them from the TS. As a difference with respect from the original TS model and in order to deal with large delay tuples, a leasing time was introduced as part of the tuple definition. This leasing time is defined when the tuple is created and it provides a way of telling the TS if a message is still relevant according to the tuple creator. For this thesis we included the notion of a tuple owner. This owner is intended to be a unique identifier of the creator of the tuple. Using the owner identifier we can retrieve tuples that match the ones we published without extracting those as it makes little sense to retrieve our own tuples. Also, tuples will be immutable so there is no available method for modifying them. Listing 3.2 shows the interface we provide for the tuple definition and available methods.

First, the owner method returns the identifier of the owner or author of the tuple. Second, the match method compare the actual tuple with the given one and returns the resulting tuple after the match. The matching procedure is

```java
public interface ITuple {

    UUID owner();

    Optional<ITuple> match(ITuple tuple);

    IField get(int position);

    int length();

    long leasing();

}
```

Listing 3.2: ITuple interface

the same discussed in Chapter 2. For this method we wrapped the resulting tuple in an optional in case there is no match so we can return an empty optional instead of a null. Third, we have a length method for returning the number of fields in the tuple. Forth, we have a get method that receives some integer indicating the field we want to access in the tuple. Notice that the position argument is 0-based so it must be between 0 and length() − 1. Finally, there is a leasing method for returning the leasing or expiration time of the tuple.

3.2.3 Field Interface

The fields are the objects from which the tuples are composed. As we discussed on the previous chapter, fields can be of two types: actuals and formals. In the case of being an actual, developers will be able to extract their corresponding value. Fields are also immutable, so again there is no available method for modifying them. Listing 3.3 shows the interface we provide for the tuple definition and available methods.

As we can see, the field interface has a generic placeholder. This is because a field can wrap any kind of element. The interface provides methods for verifying whether a field is either a formal or an actual. Definitions of formal and actual fields were discussed in Chapter 2. In case the field is an actual, there is a method for retrieving the element that the field is holding. Also,

```java
public interface IField<T> {

    boolean isFormal();

    boolean isActual();

    Class<T> type();

    T element();

}
```
Listing 3.3: IField interface

for both actual and formal fields, there is a method for returning the type of the field. This type is returned as a generic `Class` object with the same placeholder as the field interface.

3.3 Routing Layer

The routing layer is mainly an abstraction between the raw communication of data between devices and the tuples in the Tuple Space. The routing process wraps information to exchange in the form of messages. In the following subsections we will discuss each of the parts that compose this layer.

3.3.1 Routing Interface

The routing interface is the main part of the routing layer. It is also the entry point for communication with both the Tuple Space layer and the communication layer. Listing 3.4 shows the code for this interface.

Notice that the interface is generic so it is going to wrap the type of elements given by the implementation. The first method the interface provides is an ID that identifies each device from the routing layer. Using this ID the devices can know about each other when they are exchanging messages. The `add` and `remove` methods receive an element, wrap it into a message and store it for

```java
public interface IRouting<T> {

    UUID id();

    void add(T element);

    void remove(T element);

    List<UUID> messagesIds();

    List<IMessage<T>> sendMessages(List<UUID> messagesIds);

    List<UUID> requestMessages(List<UUID> messagesIds);

    void receiveMessages(List<IMessage<T>> messages);

    boolean shouldCommunicate(UUID hostId);

}
```

Listing 3.4: IRouting interface

further exchange. The messagesIds method returns a list with the identifiers of the messages stored by the layer

The following methods are intended for communication with other hosts. The sendMessages method receives a list of requested messages IDs (probably received from other host) and returns a list with the messages corresponding with each of the IDs given. Notice that it can be the case that some of the requested messages do not exist in the layer so they will not be returned. On the other hand, the requestMessages method receives a list of messages identifiers that possibly some host have and returns a list with the messages identifiers that the layer is missing from the given list. The receiveMessages method is intended to be used from the communication layer for adding incoming messages to the layer and add the elements to the Tuple Space layer. Finallly, the shouldCommunicate method determines if communication with some specific node (determined by its identifier) should be performed. Depending on the implementation, all nodes could be allowed to communicate, but other ideas could explore blacklisting or avoiding recently encountered hosts.

3.3.2 Message Interface

A message is the basic unit that the routing protocol understands. A message works as an envelope for wrapping higher level objects so the routing layer can work with them. For example, in this thesis we use them for wrapping tuples from the Tuple Space layer. The routing layer may hold a message for each element in the upper layer in such a way that it can add or remove messages at the same time that tuples are added or removed. It is worth mentioning that the message can hold any kind of object so the interface will have a generic placeholder in order to define the type of the wrapped element. Listing 3.5 shows the interface we provide for messages.

```java
public interface IMessage<T> {

    UUID id();

    int hopCount();

    T element();

}
```

Listing 3.5: IMessage interface

The methods for this interface work as getters for the main attributes of the message in a pretty straight-forward manner. The id method returns a UUID object denoting a unique identifier for each message in the system. The hopCount method returns the number of hops that the message can jump from node to node. A hop count of 0 means that the message should not be exchanged anymore. Finally, the element method returns the element contained or wrapped by the message.

3.4 Communication Layer

The communication layer is the lowest layer in the framework stack. This layer has a device dependent implementation as it will call directly the provided APIs for communication. For example, Android devices would call a different set of APIs than laptop devices. This layer will communicate only with the routing layer and it does not have an interface as it is not called by any piece of code. For this thesis we focused on Android applications so the communication layer we implemented uses the APIs provided by the Android SDK. More specifically, we used the Wi-Fi Direct APIs for Peer-2-Peer device communication. Using Wi-Fi Direct, we were able to transfer information without requiring a hotspot.

Chapter 4

Evaluation

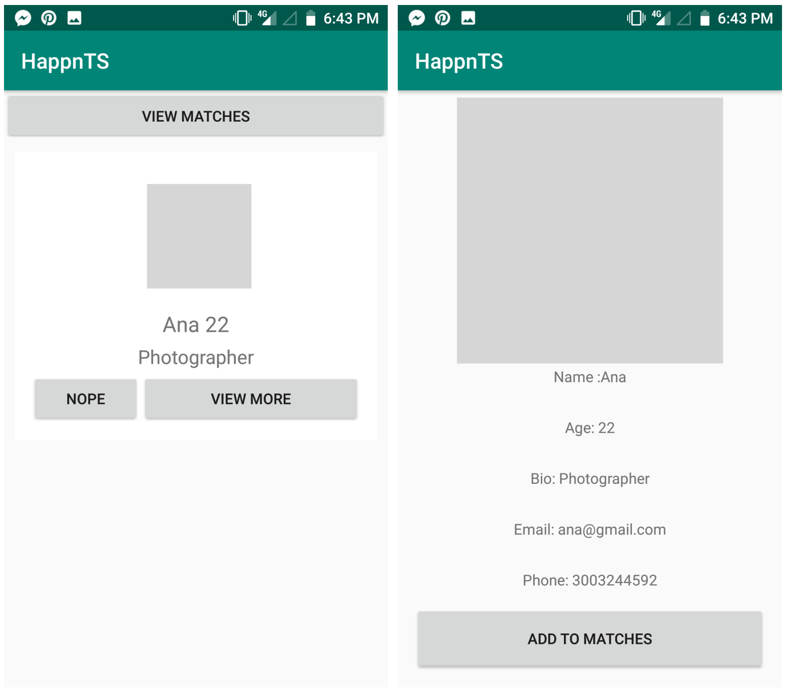
In this chapter we show two simple applications we made in order to show the usage of the framework. In each of the following sections, we briefly describe each of the applications and discuss some of their implementation details.

4.1 HappnTS

4.1.1 Description

HappnTS is a small clone of a popular application called Happn[1]. Happn is a location based dating app that shows you people you have crossed paths before so you can start a conversation with them. HappnTS is a variation in which you can get information about people you crossed nearby and either ignore them or add them to your list of matches. The information profile each person publishes consists of a name, a short bio, a profile picture, an age, an email, and a phone number.

[1] `https://www.happn.com/en/`



Ana 22

Photographer

Name :Ana

Age: 22

Bio: Photographer

Email: ana@gmail.com

Phone: 3003244592

(a) Main Activity (b) View Profile Activity

Figure 4.1: HappnTS application preview.

```java
private void publishMyself() {;
    Person person = new Person(
        "Juan", "Coder", 23, profilePicBase64,
        "jcbages@outlook.com", 3003244592L);

    ITupleSpace TS = TupleSpace.getInstance();
    TS.out(person.toTuple(PeopleData.OWNER_ID));
}
```

Listing 4.1: Person profile publish

4.1.2 Implementation

The main differentiation of HappnTS is that it works as an infrastructure-less network. Passing of data occurs directly between devices when people are close enough so communication can happen. The way we achieve this is by publishing a person's profile tuple for each instance of the application on the device's local Tuple Space as we show in the code snippet below.

In the first step we created a Person object with our profile information. Then, we instantiated the Tuple Space and added that tuple with a previously defined owner identifier using the out operation. Listing 4.1 shows how we did this.

As soon as the application publishes its own profile information tuple, it starts searching for other people indefinitely using the in operation. In order to do this we decided to use the AsyncTask class provided by Android so we can wait synchronously for new arriving tuples without blocking the main thread. Listing 4.2 shows a simplified version of the main logic behind this search process.

Notice that every time we find some matching tuple, we have to go to the main thread in order to update the UI of the application. Just after that, the AsyncTask is restarted so that it can search for more matches. In order to abstract the conversion between local objects and tuples, the Person class implements a method toTuple for converting an object to a tuple, a method outTuple for creating an object from a tuple, and a method inTuple for creating a "search" tuple that matches other people profile tuple as shown in Listing 4.2. These abstractions correspond to the API of the Tuple Space layer.

```java
public class PeopleAsyncTask extends AsyncTask<Void, Void,
    Person> {
    ...
    @Override protected Person doInBackground(Void... args) {
        ITuple inTuple = Person.inTuple(PeopleData.OWNER_ID);
        Future<ITuple> future = TS.in(inTuple);
        ITuple personTuple = future.get();
        return Person.outTuple(personTuple);
    }

    @Override protected void onPostExecute(Person person) {
        PeopleData.addPerson(person);
        this.mainActivity.get().restartAsyncTask();
    }
}
```

Listing 4.2: Profile search AsyncTask

```java
public class Person {
    ...
    private UUID id;
    private int age;
    private long phone;
    private String name, shortBio, profilePic, email;
    ...
    public ITuple toTuple(UUID owner) {
        long leasingTime = Long.MAX_VALUE;
        return new Tuple(owner, leasingTime,
            new Field<>(String.class, TAG), ...);
    }

    public static ITuple outTuple(UUID owner) {
        return new Tuple(
            owner, new Field<>(String.class, TAG), ...);
    }

    public static Person inTuple(ITuple tuple) {
        return new Person((UUID) tuple.get(ID_POS).element(),
            (String) tuple.get(NAME_POS).element(), ...);
    }
}
```

Listing 4.3: Person class tuple methods

```
/**
 * Set the default hop count to 1
 */
Message.setDefaultHopCount(1);
...
```

Listing 4.4: Hop count modification

Something important about the application is that we only want information about the people we crossed and not from the people that others crossed. That is, we do not want our profiles to jump more than once. In order to achieve this, we set the default message hop count to 1. This way, as soon as our profile is published it will have an initial hop count of 1. But as soon as it reaches other devices it will arrive with a hop count of 0 so it is not going to be transmitted from those devices further to other nodes. The only modification required for this is changing a parameter in the Message class implementation for the routing layer as Listing 4.4 shows.

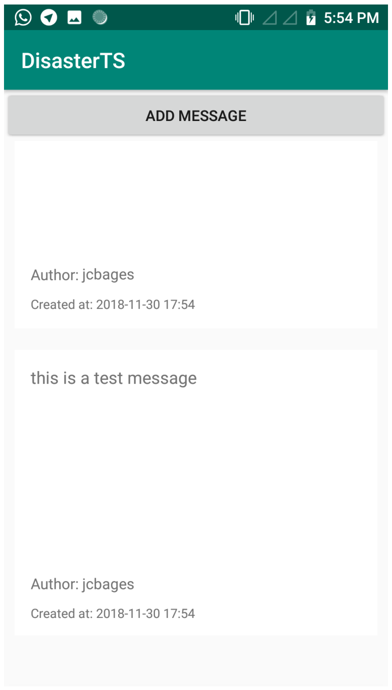
The rest of the application code is more concerned with specific logic not related to the communication between devices. As we can see, using our framework provided a simple and abstract way of coding this communication interactions just by thinking of inserting and retrieving tuples locally. The application source code is available at the FLAGLAB org[2].

4.2 DisasterTS

4.2.1 Description

DisasterTS is a simple application for communication in case of natural disasters. Natural disasters such as earthquakes can damage the existing infrastructure leaving victims unable to communicate between them. The purpose of this application is to send plain text messages with no more than 200 characters or less that spread in an infrastructure-less environment across all possible devices. By using DisasterTS victims can report their location, status, and other important information they need or want to share.

[2]https://github.com/FLAGlab/communication-in-dmanets-using-tuple-spaces



Author: jcbages

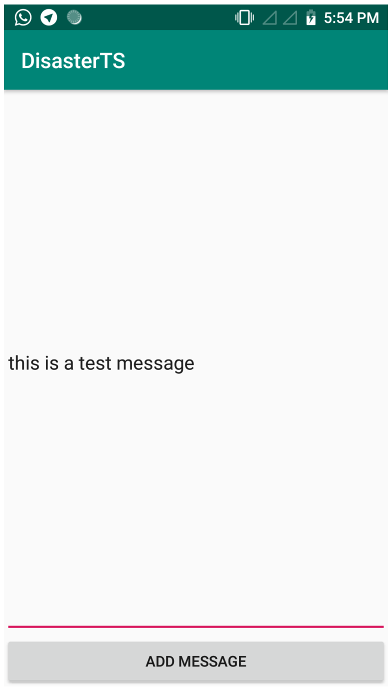
Created at: 2018-11-30 17:54

this is a test message

Author: jcbages

Created at: 2018-11-30 17:54

(a) Main Activity



this is a test message

(b) Add Message Activity

Figure 4.2: DisasterTS application preview.

```java
public class Message {
    ...
    private UUID id;
    private String text, author;
    private Date date;
    ...
    public ITuple toTuple(UUID owner) { ... }
    public static ITuple outTuple(UUID owner) {...}
    public static Message inTuple(ITuple tuple) {...}
}
```

Listing 4.5: Message class

4.2.2 Implementation

The basic unit of information that we exchange in this application is the Message object. A message consists of a text with a maximum length of 200 characters, an author that is unique for each device, and a creation date. As with the HappnTS application in the previous section, we add methods for converting a Message into a tuple, generating a search tuple that match valid messages, and converting a tuple into a Message. The reason for these methods is to reduce the amount of Tuple Space related code in the main logic. This is the same idea that we used in the Person object for the HappnTS application. Listing 4.5 shows an overview of the Message class.

As soon as the application is initialized, it starts listening to new messages. The way we handle this is very similar to the way we retrieve new matches in the HappnTS application. That is, we start a new AsyncTask that will wait synchronously for the future returned by the Tuple Space in operation to resolve. As soon as we retrieve a tuple, we will insert it again so it can be transmitted to other devices. In this case the user publishing the tuple is defined as the owner of the tuple so as not to read self-published message again. Listing 4.6 shows a simplified view of the AsyncTask class.

As disasters messages may contain critical information, we want them to spread as much as possible across every victim device. The way we configure this in our framework is by increasing the hop count parameter so that messages can reach more Tuple Spaces. Listing 4.7 shows how we configured this constant for the DisasterTS application.

```java
public class MessageAsyncTask extends AsyncTask<Void, Void,
    Message> {
     ...
     @Override protected Message doInBackground(Void... voids) {
          ITuple inTuple = Message.inTuple(MessagesData.OWNER_ID);
          Future<ITuple> future = TS.in(inTuple);
          ITuple messageTuple = future.get();
          return Message.outTuple(messageTuple);
     }

     @Override protected void onPostExecute(Message message) {
          // publish it again with me as an owner
          ITupleSpace TS = TupleSpace.getInstance();
          ITuple tuple = message.toTuple(MessagesData.OWNER_ID);
          TS.out(tuple);

          MessagesData.addMessage(message, false);
          this.mainActivity.get().restartMessageAsyncTask();
     }
}
```

Listing 4.6: Messages search AsyncTask

```java
/**
 * Set the default hop count to 20
 */
Message.setDefaultHopCount(20);
...
```

Listing 4.7: Hop count tuning

The rest of the code is related with application specific logic. Again, our framework provided an easy way for transmitting Message objects without thinking about how the underlying communication works. The application source code is available at the FLAGLAB org.[3]

[3]https://github.com/FLAGlab/communication-in-dmanets-using-tuple-spaces

Chapter 5

Conclusion

5.1 Conclusions

In this thesis we have introduced a framework for developing applications that work on infrastructure-less environments. Using this framework we are able to abstract the communication between nodes by taking advantage of the simplicity of the Tuple Space model. Additionally proposed, the layered architecture allows us to divide responsibilities among the different layers in such a way that changing the implementation of one layer did not require significant changes to the other layers.

5.2 Future Work

The framework we introduced is more a proof of concept rather than a production ready piece of code. The following are some of the areas in which we think progress could be done in order to make the framework more suitable for real life scenarios:

- Efficient tuple matching is a big area for improvement as tuples can have arbitrary length and values. An efficient algorithm for matching needs to increase the performance of the applications that use the framework while taking into account resource consumption as nodes are limited in hardware.

- Mechanisms for transferring messages between devices can be made more resilient to eventual connectivity between hosts. As data transfer can be interrupted unexpectedly, nodes need to be prepared for this scenarios. The communication layer would be the target in this case.
- Resiliency is also needed in the case nodes fail. This can happen because of the device's limited battery or because of an application crash. In both cases the Tuple Space need a secondary storage backup like a log so that its can go back to its original state after the node recovers from a failure.
- Even though we applied our framework using only Android applications, it can be easily extended for other devices such as laptops. In order to do this, the communication layer would need to be implemented for the target device.

Bibliography

[1] Elizabeth M Daly and Mads Haahr. The challenges of disconnected delay-tolerant manets. *Ad Hoc Networks*, 8(2):241–250, 2010.

[2] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34. ACM, 2003.

[3] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[4] Abdulkader Benchi, Pascale Launay, and Frédéric Guidec. A p2p tuple space implementation for disconnected manets. *Peer-to-Peer Networking and Applications*, 8(1):87–102, 2015.

[5] John P D'Angelo and Douglas B West. Mathematical thinking. *Problem Solving and Proofs*, 1997.

[6] Ian Foster. *Designing and building parallel programs*, volume 78. Addison Wesley Publishing Company Reading, 1995.

[7] Amin Vahdat, David Becker, et al. Epidemic routing for partially connected ad hoc networks. 2000.