Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Conway's Game of Life in a Distributed Adaptive System

Ingegneria dei Sistemi Software Adattativi Complessi

Editors:
Francesco Cozzolino
Marina Londei
Luca Longobardi

Introduction

The project presented in this report consisted in implementing the *"Game of life"* for a mobile environment, in which the interaction between devices is started by *pinching* each other and, after that, the game continues on the grid formed by the connected group.
This idea leads to the rearrangement of devices and their calculus, and it can be used for various applications. Our work led us to face with different problems, most of them derived from one of the main arguments of this course, that is the complexity of distributed systems and their management.
A system can be defined complex when, for example, we have to deal with different components, potentially heterogeneous, and when there's a decentralized and distributed environment.
During our work, we had to decide which way was the best one for managing the interaction between devices, and our decision ended up in being the use of *Message Oriented Middleware* (RabbitMQ, in our specific case).
Our project, so, strictly embrace the idea of ***Internet of Things***.
The report follows this structure: first, we give an overview of what the distributed communication means, and all the challenges it gives; second, we explain what a Message Oriented Middleware (from here referred to as "MoM") is and why its use is fundamental in a distributed environment; third, we enter the core of our project, explaining first of all the idea behind the *pinch gesture*, and then explaining in details how the system works and how we manage all the aspects of its realization.

Contents

Chapter 1

Distributed Communication in complex systems

1.1 On the meaning and nature of complex system

A complex system can be described as a *system composed by sub - components intertwined together that interact, highly sensitive to change.*
These systems are the emerging ones nowadays, so it's very important to understand how we can build and manage them in a correct way.
These systems are:

- Distributed - since they are composed by physically different machines
- Mobile - since the single components are not fixed and can move through space
- Open and dynamic - since the number of devices involved can change during time

Another important concept is the one of the **adaptivity** of a system, that is the ability to change its behaviour depending on circumstances. We want

our system to be highly adaptive, and to reach this goal we need the components to be able to *monitor* data exchanged, *detect* symptoms and *deciding* changes consequently, then *acting*, applying changes and producing an effect.

1.2 Distributed Communication

To be able to create a robust and functioning complex system, able to respect the requirements of the project, we have to face the problem of *integration between entities*.
In a distributed system as the one we are dealing with, we cannot rely on classic methods of interaction like the local method invocation, since we're working on a remote context and there's no local communication.
In this kind of system, we rely on network that is by its nature unreliable and slow; also, it has to be considered that our system is subjected to change over time.
There must be no **tight coupling**, where **coupling** means how much we can assume about the other components. Since each component potentially differs for OS, and also applications that communicate are different between each other, we can't make any assumption on the nature of the other components. In a tight-coupled environment we send information directly to a specific machine, and this creates a dependency that we can't afford to have in a distributed environment: what if the address of the machine changes? What if the same information has to be sent to other machines? We would have to change everything, every time something like that occurs (and in our kind of systems, this happens too many times to be managed this way).

Instead of sending informations to a specific machine, we should send them to an addressable channel (a logical channel), so that both sender and receiver can interact, without knowing each other's identity.
Another dependency that we want to avoid is the *time dependency*, that

is connected to the synchronous interaction. The interaction between two entities is time dependent if these two components have to be active and connected in the same moment to make the communication possible; if this prerequisite isn't satisfied, then the communication is not possible.
In a distributed and complex environment, it is impossible (and useless) to ensure that the entities that have to communicate will be active in the same moment, so we need a communication solution that will work without this assumption; in other words, we want to ensure ***asynchronous communication***, with channels that queues up sent requests until the network and the receiving system are ready.
An integration solution that ensures ***loosely coupling*** communication consists in using a *message based integration*; basic elements of this solution are:

- Communication channel: to move information from one app (or component, or system) to another
- Message: data to be exchanged
- Translation: to bridge different internal data formats
- Routing: to move the data to proper address/location
- System Management Function: to monitor data flow and for error reporting

Message based integration is an asynchronous interaction model that guarantee persistence of information and loose coupling between entities.

Chapter 2

Message Oriented Middleware

A message oriented middleware (also referred to as MOM) is a messaging service that enables communication among different applications, devices, and, in general, systems. The word "*middleware*" refers to an infrastructure that provides messaging capabilities, mediating between entities and creating a uniform layer for communicating, guaranteeing interoperability.
In addiction to the elements of a general message based integration, the middleware offers:

- Pipes and filters - to process messages
- Endpoint - to connect and application to the messaging channel

MOM is an evolution of traditional RPC (Remote Procedure Call), that required simultaneous availability of participants in the communication; it supports communication of process that are not (necessarily) executing at the same time, making of use of *message queues*.
The middleware offers messages management, error reporting, persistence and a middle layer that decouples participants.
It guarantees *reliability* and *scalability*, although there is complexity in the management of asynchronous interactions. Also, since the coordination is based on message passing, it has to ensure the correct order of events; to do that, MOM offers a queuing layer to store messages.

The middleware offers two messaging models:

- **Point to Point** : asynchronous message passing between software entities
- **Publish - Subscribe**: one-to-many or many-to-many. This model disseminates information: the entities publish messages to a specific topic or channel

Message based interaction is mandatory for IoT systems, that are intrinsically distributed and unreliable.
One open source implementation of a MOM is *RabbitMQ*, that is the one used in this project.

Chapter 3

The Project

In this chapter we present our work and how it has been carried on, focusing on the three principal aspects of our efforts: the managing of informations between devices, the *Pinch* gesture to connect two devices and the way we managed the cell calculus.
As already said, we worked with a mobile environment, so in the context of *Internet of Things*. The IoT is "*a system of physical objects that can be discovered, monitored, controlled, or interacted with by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider internet*" [2].
This is the idea of *pervasive computing*, where entities can be smart devices, embedded devices, that interact between them and with the people.
Our project focuses on the distributed calculus between devices, without the managing of a central unit. Each device knows what it has to do, and all the interactions are managed through a Message Oriented Middleware.

Conway's "Game of Life" - Why we chose it

Before we begin, it is convenient to explain what "Game of Life" is.
It has been devised by mathematician John Conway, and it shows the evolution of an initial population of cells; beside choosing an initial configura-

tion, one interacts with the "game" by simply observing how the population evolves.
The universe considered in this game is a grid of square cells, each of which can be either in a *dead* state or *alive* state. Whether a cell will be dead or alive in the next generation, depends on the neighbours of each cell, which are the cells horizontally, vertically or diagonally adjacent to the one considered. Each step, for each cell, one of the following transitions can occur:

- Any live cell with fewer than three live neighbours dies
- Any live cell with three live neighbours survives to the next generation
- Any live cell with more than three live neighbours dies
- Any dead cell with three live neighbours comes alive

There are lots of possible variations to this game; we chose to follow the rules of the original one.

The reason why we decided to use this specific simulation is because it is one of the first and simpler programs we come across, and yet it fits perfectly to the idea of distributed calculus we wanted to show.
Starting with a single device, we wanted to extend the game to a group of interacting devices, considering the universe of the game as composed by the single grids in each device. The population, so, evolves in a grid that gradually enlarges each time a device is connected to one another.
We chose to connect two devices by *pinching* them, that consists in a swiping gesture on each device, making forefinger and thumb one on the display of each device meet each other. This gesture is very simple, and gives the idea of connection between the devices.
Behind this, there's a Message Oriented Middleware taking care of messages exchanged between the couples of devices, containing informations about how

the connection and the game itself.
Each device is responsible solely for itself and the information it has to send to the other device connected to it, and there isn't a central unit to control the flow of the game.

3.1 Assumptions

For our project we made different assumptions.
First of all, it is possible to pinch devices only when the game is in pause mode; this is done both for a reason of logic but especially to avoid inconsistency in the calculus of generations.
Second, the pinch between devices can be done one pair at a time: this means that it's not possible to pinch two or more couples of devices at the same time. This is because we have to avoid overlapping of messages, since we work on a broadcast channel, and it would be very difficult to recognize and isolate the couple of devices, recognizing which one was pinched with another one.
Third, when swiping on to devices to pinch them, the lines of the grid have to be aligned, both for a reason of correct indices calculus and for aesthetics in game.
Another thing is that we assume that the server is always online. In fact, if a device disconnects, it is possible to handle it because it is immediate to know if the connection with the server is still open; on the contrary, if the server itself is not reachable any more, it's not possible to know it in due time, because it also could be a slow network issue. In case the server is not reachable, the app will eventually stop (after some time); it is sufficient to tap "Wait" when app isn't responding, lift the device and start the game again locally.
Last, the taps on the screen do not have to be concurrent: this means that we have to perform a single *start* or a single *stop* on our group of connected

devices (and not on two devices at the same time), waiting for the operation to be sensed by all devices (since we have to take into consideration the propagation time of the messages).

Besides, these assumptions were made also for reasons of time and complexity, and because we can guarantee consistency on the single device and not on the entire group, that could be arbitrarily wide.

3.2 Pinch

The idea of using this kind of gesture to connect devices came from an article of Takashi Ohta and Jun Tanaka from Tokyo University of Technology [1]: in their paper they present this gesture as a new and more intuitive interface to connect devices, without using sensors attached to them or having to register each device manually. This method can be used, of course, only taking for granted that the devices have a touch screen.
This action can be done by, as already said, juxtaposing devices and then making forefinger and thumb swiping them until they meet each other; this gesture will then trigger exchange of information and rearrangement of the game and its visualization.
By assuming that the gesture is made with the same hand (and so that the two fingers move on the same straight line), we can deduce the respective position of the devices, making possible to share correct informations; instead, if a user swipes with two hands and in directions that are not on the same line and not opposite, there is no safe way to determine what is the exact position of the devices.
To make it possible for one device to connect to the others, it has to be connected to the same network, and having the application running. When one device notices that a swipe has occurred, it sends informations to the others connected to its network; this motion is expected to occur at two devices simultaneously, so we use the informations (in particular, the timestamp rel-

ative to the action) to identify the pair and connect the devices. Details about what and how the information is managed will be explained in the next section.
To disconnect a device, a simple gesture of shaking is used.

3.3 Devices' Communication through RabbitMQ

For our project, we decided to implement communication between devices using RabbitMQ as middleware. In the following sections we explain what RabbitMQ is and does in detail, and then how we used it for our purpose.

3.3.1 What is RabbitMQ

RabbitMQ is an open source Message Oriented Middleware, giving us a common platform to send and receive messages. It intermediates for the entities, and offers the possibility to implements and use different patterns of message exchange.
It separates sending and receiving data, decoupling applications.
In RabbitMQ there is a *producer* that is essentially the *sender* of a message; symmetrically, there is a *consumer* that is the one who waits to receive messages. A fundamental concept is the ***queue***, that is where the messages are stored. It can be thought as a message buffer, used by producers to send messages and by consumers to retrieve them. RabbitMQ acts as a *broker*, taking care of all the queues; consumers, producers and broker does not have to reside on the same host. Since we're working on a distributed environment, they clearly don't.
As simple example of how this middleware works, we can think about a producer that sends a message to a consumer and then disconnects; the steps for the producer are:

1. *Create a connection to a server* - we create a "Connection Factory", setting the broker to a machine (ex: localhost or IP address of the

machine), the we create the *channel*, in which we will create the queue we need

2. *Declare a queue to send* - to publish a message, we have to create a queue to send to, specifying the name
3. *Publish message* - using the channel, we can now publish a message as byte array
4. *Close channel and connection*

The consumer acts different from the producer: it will keep running to listen for messages.
The initial steps (up until the declaring of the queue) are the same of the producer's: we create a connection to the server, create the channel and then declare the queue, that of course has to have the same name of the queue the producer is sending message to.
Now we have to tell the server to deliver us the messages from that queue: to do that, we use a callback as an object that will buffer messages until we use them.
Any time a message is received on the queue, this callback can handle it and the consumer will take some actions.
Beyond this simple example, RabbitMQ offers the possibility to set up different ways of communication; as an example, we can implement a publish/-subscribe mechanism, adding the concept of *exchange*, that receives messages from producers and on the other side it send them to the queues. The responsibility about what it has to be done with the message is not of the producer, but of the exchange, that knows if, for example, the message should be sent to a particular queue or multiple queues.
There are different types of exchange, each of them realizes a different behaviour; one of them (that we used in our project) is the "fanout" type, which simply broadcasts all the messages it receives to all the queues it knows.
To bind an exchange to a queue we create a relationship called *binding*, that will let the exchange send messages to our queue.

Moving one step further, another thing that can be done with RabbitMQ is subscribing only to a subset of messages; this is called *routing*. When binding an exchange to a queue, we can specify a *routing key parameter*; this way, the queue will be interested only on the specified kind of messages.

3.3.2 Communication through RabbitMQ

In this section we will explain how RabbitMQ and its features were used and adapted in our project to realize the behaviour we wanted to show.
Before we begin, it is necessary, for a better understanding, to explain briefly the structure of the classes that have major roles in communication. These are:

- **RabbitMQ.java**: this class contains all the methods used for connecting to the server and send/retrieve messages from queues. It encapsulate the main features that the middleware gives us, making it possible to create channels and declare queues, subscribe to one of them, send messages to a specific queue, add a listener to that queue so to handle the delivery of messages (through a callback) and close a connection.
- **Handler.java**: this class handles the delivery of messages for each device, taking different actions with regard to the type of message received. It manages the pinch informations when a swipe is detected and, consequently, binds the two devices by creating the queues for sending and receiving; it also manages the closing of connection between two devices and the logic behind the start - stop messages for the game. Moreover, it is the class that sends and receives the cells exchanged between one device and another.

First connection

When opening the application, the first thing we do is simply connect the device to the server. The IP address of the server can be changed when needed modifying a txt file on the device.

This step is fundamental, since if the devices are not connected to the same server, the communication won't establish.
Also, we also add the exchange and we bind the queue to send and receive broadcast messages relative to the swipe (as explained below).

Pinch connection

When a swipe is perceived, the information is sent to all the other devices on which the app is running. To do that, we use the "Publish/Subscribe" pattern. This simple pattern allows to send the same message to anyone who is interested.
In our case, we are interested both in sending and receiving the messages concerning the swipe gesture, so a device is a publisher and a subscriber at the same time.
After connecting to RabbitMQ's server, the application declares the *exchange* to send the messages to. An exchange allows to route the messages to the subscribers.
As we are interested to receive the messages from other devices, we create a queue to do that, and bind it to the exchange. This operation is essential for receiving the messages, since it tells the exchange who to send the messages to.

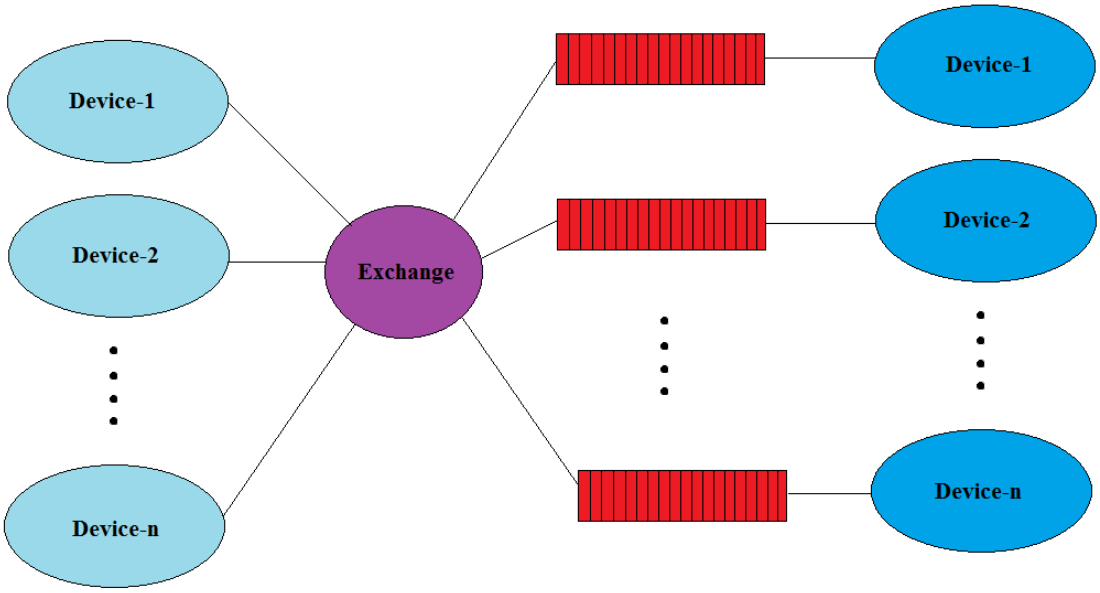

Figure 3.1: Broadcasting of a swipe message

As we see from the figure, when a device sends a swipe message, it receives that message too, so it is necessary to filter the incoming messages. When a swipe is sensed, the application sends the following informations:

- Ip address of the device
- Direction of the swipe
- Coordinates in pixel where the swipe was done
- Timestamp when the swipe occurred
- Width and height of the device in inches
- DPI of the device

When a device receives this kind of message, it retrieves the informations about the last swipe that user performed on it and checks if it's occurred in a moment close enough to the swipe that occurred on the other device. If the the difference between timestamps of the two swipes is smaller than a threshold, the application adds the other device on a personal map of connected devices and, thanks to the informations received, it calculates how many and which cells it will have to send/receive from the other device. Furthermore, both devices declare to RabbitMQ's server the queues necessary for the communication between them. The queues need to be unique: to guarantee this uniqueness, we associate them a name obtained by the concatenation of the IP addresses of the two devices.

Management of generations

When the game is running on a device connected with one other or more, the management of the generations is slightly different from the simple case where the device is running alone, since we have to face all the problems linked to the synchronization and sending of messages.
The dynamic of how the generations are managed is encapsulated in the class "*CalculateGeneration*"; here we explain how it works:

1. The device sends the correspondent cells to each one of its neighbours (the cells were calculated at the moment of pinch with that device; details about the calculus and setting of the cells will be explained in section *Cell Calculus*), setting the flags that indicate whether the device has sent the cells to the other to TRUE

2. The device waits until it is ready to go on; this happens when it has all the required cells to calculate the next generation. The cells received are stored in a list, one for each device the current one is paired with; in details, each one is a list of lists, where an element represents the list of cells of a generation. Each time a device has to calculate the next generation, it removes the first element of the list for each connected

device, since it represents the oldest generation that it didn't calculate. This mechanism prevents the devices from de-synchronizing from each other, even in a case when, for any reason, the stop message is not received from all the devices in the network; in fact, a device will not go on with the calculus until it has at least one generation available from each list (that is, from each device)

3. The flags indicating whether the device has sent the cells to the others are set to FALSE

4. The device sets the received cells

5. Using the informations received and the cells set, the device calculates the next generation

6. Grid is redrew

7. "Ghost" cells are reset (= set to false, ready to be set again according to the informations of the other device)

8. The routine restarts

All the messages are sent by the Handler on the queues that were created at the moment of pinch between two devices.

Start - stop management

Beyond the normal routine of the game, it is possible for the user to stop the game at any time.
If the game routine is stopped, the device on which the user has double tapped to stop sends to all its neighbours a message of "pause", that the other devices will then forward to their neighbours, and so on.
If the game is in a state of "pause" and the user double taps the screen, the game is started again: the device on which the double tap has been performed sends a message of type "start" to all it neighbours, that will eventually

forward the same message to their connected devices. The computation of generations can start again: the device sends the cells to all the neighbours that didn't send them to previously (in fact, it can happen that the "stop" command is done when the device is still sending its cells to the others), then keeps going with the calculus, with the same logic we explained earlier.
All the messages are sent on the queues that were created at the moment of the pinch between two devices.

Disconnection management

To detach a device from another one or from a group, it's enough to lift it or close the application. With the accelerometer it's possible to detect rough movement and, when it occurs, the application removes all devices from the map of connected devices and sends a message of closure before closing the channel with them. This message is sent through the queue created at the moment of the pinch.
When a device receive this message, it simply closes the correspondent channel and removes the sender from the map of connected devices. This message is used to avoid an endless wait for receiving the cells from a disconnected device when the game is still running.
If the user closes the application, furthermore, the connection between the device and the RabbitMQ's server is also closed.

3.4 Cell Calculus

Since we don't maintain a central representation of the grid, but we consider the result grid after the pinch as the union of the two grids of the devices (or, if one of them was already pinched to someone else, the union of other unions), we had to find a way to send and store information about the enlarged grid, so to evolve correctly the cells on the borders of the devices, without having to rely on a central representation or a coordinator.
We thought that the best solution was to consider the matrix that represents

our grid with two additional rows and two additional columns to simulate the borders of the (potentials) juxtaposed devices.
Those two rows and two columns are initially set to the dead status, as if the device is not connected to anyone: this has the same effect as if we considered the matrix without the additional rows and columns, so it does not affect the calculus if the device is on its own.
When a device is pinched with another one, we send and receive the information about which of the four additional rows or columns we have to consider, and which indices correspondent to the cells we expect to receive in the next generation.
To carry on this calculus, we have to go through three main steps:

1. Calculate the respective positions and orientations of the pinching devices
2. Calculate the portion of screen in contact with the other device and the indices of the cells to be sent
3. Preparation of cells to be sent

1 - Position and orientation

The two pinching devices can be in different positions with respect to each other, and this has to be calculated before any further step, since the calculus differs with regard to their positions; an example is shown in the figure below:

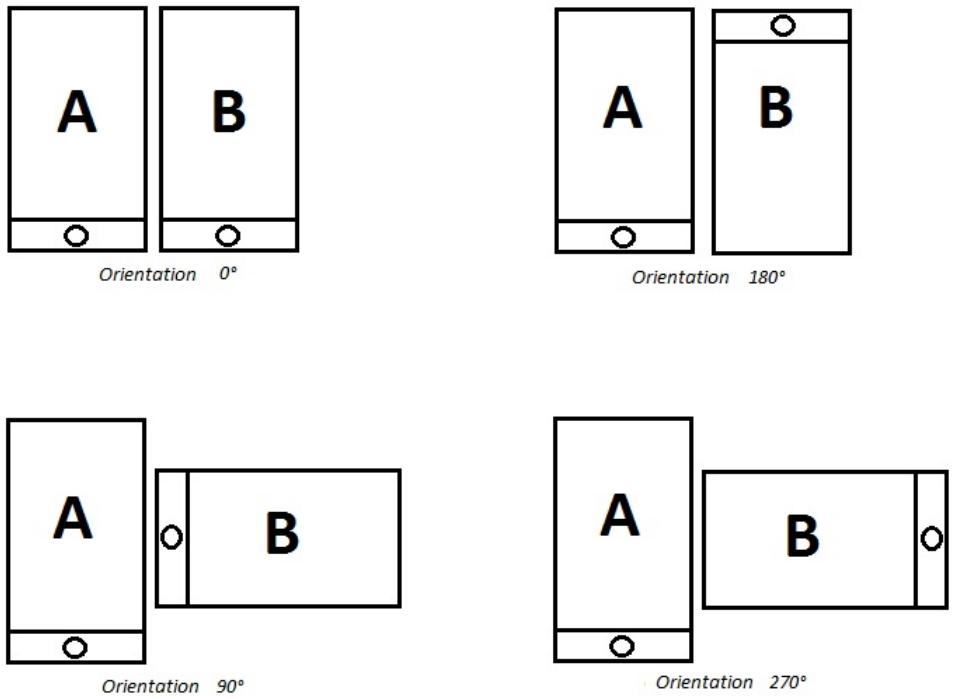

Figure 3.2: 4 different devices orientations - one side only

Considering the device **A**, the other device pinching with him can be in four different orientations. This, of course, has to be considered on the four sides of the device, leading to 4x4 = 16 combinations of respective positions. Since the size and the drawing of the grid does not change whether the device is in portrait or landscape, we chose to lock the app in portrait mode, since this way we avoided to consider (uselessly) 16 more combinations for the calculus.
To calculate the effective respective positions we used the informations about

the swipe direction, both of the device considered and the other one, as can be seen in the example:

```
if(myDir.equals(PinchInfo.Direction.RIGHT)){
    if(dir.equals(PinchInfo.Direction.LEFT)){
        this.orientation = 0;
    } else if(dir.equals(PinchInfo.Direction.RIGHT)){
        this.orientation = 180;
    } else if(dir.equals(PinchInfo.Direction.DOWN)){
        this.orientation = 90;
    } else if(dir.equals(PinchInfo.Direction.UP)){
          this.orientation = 270;
    }
}
```

myDir represents the direction of the swipe of the current device, while *dir* is the direction of the swipe of the other device pinching. Each combination gives us information about what the orientation of the other device is; this will then be used to differentiate the calculus for the cells.
As further explanation, have a look at the picture below:

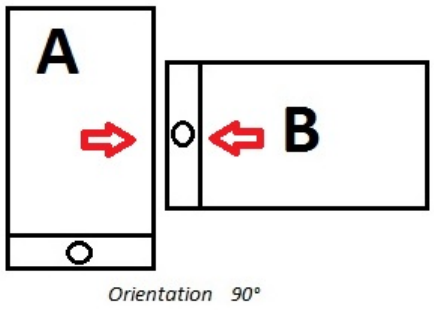

Figure 3.3: Example of orientation calculus

For the device **A** its swipe direction is ***RIGHT***, and it will receive from

B that its swipe direction is ***DOWN***; knowing this, we can conclude that the respective positioning of the two devices is in fact the one shown in the picture, so the orientation of the other device (speaking for device A) will be of ***90***°.
The same considerations are made for all of the 16 cases identified.

2 - Portion of the screen in contact and cells indices

Once calculated the orientation of the devices, we can proceed to the calculus of the portion of the screen that is in contact with the other device. This calculus is very important, because we have to know the exact cells of the grid that are to be sent to the other device (and, symmetrically, to be received).
It has to be considered that:

- The devices can be of different size: one of them could be a tablet and the other one a smartphone (or, simply, two smartphones of different screen size)
- The devices, even if with the same size, may not be juxtaposed for their entire width or height, but there can be just a portion of the screen that is really in contact with the other

Having in mind this, we had to find a way to determine exactly how much and what part of the screen is effectively in contact.
To do that, we made use of the coordinates' informations relative to the swipe of both devices: we considered, for each device, the point of the swipe as the point by which the screen is split in two parts, and we calculated the length of these two parts.
Of course, we have to consider also the length of the two parts of the other device and, for each correspondent part, choose the shorter one, because we don't want to send more cells than necessary.
The figure below explains this issue:

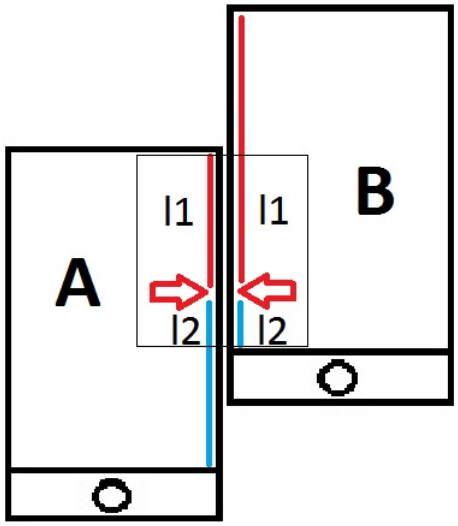

Figure 3.4: Example of calculus of portion of the screen

The portion of the screen in contact is the one inside the square.
Each of the two devices calculates (using the informations received) the length of L1 and L2, both its and of the other device, then choose the minimum value for each of the two measures: this way, we are sure that we won't send cells that are not part of the portion considered.
After this, we set the values of *first index* and *last index* of the cells that we will send; this is done by dividing the coordinate of the point in correspondence of which we detected the swipe, locating it in a cell on the screen, and adding (or removing) the minimum values calculated to identify the two indices we need.

```
int min;
if(myDir.equals(PinchInfo.Direction.RIGHT)
        || myDir.equals(PinchInfo.Direction.LEFT)){
    if(orientation == 0){
        min = Math.min((int)(myYCoord/this.cellSize),
                       (int)(yCoord/this.cellSize));
```

```
            this.indexFirstCell = (int)(myYCoord/this.cellSize)+1-min;
            min = Math.min((int)((myHeight-myYCoord)/this.cellSize)+1,
                           (int)((height - yCoord)/this.cellSize)+1);
            this.indexLastCell = (int)(myYCoord/this.cellSize) + min;
        } else if (orientation == 90){
            min = Math.min((int)(myYCoord/this.cellSize),
                           (int)((xCoord)/this.cellSize));
            this.indexFirstCell = (int)(myYCoord/this.cellSize)+1-min;
            min = Math.min((int)((myHeight-myYCoord)/this.cellSize)+1,
                          (int)((width-xCoord)/this.cellSize)+1);
            this.indexLastCell = (int)(myYCoord/this.cellSize) + min;
        } else if (orientation == 180){
            min = Math.min((int)(myYCoord/this.cellSize),
                           (int)((height-yCoord)/this.cellSize));
            this.indexFirstCell = (int)(myYCoord/this.cellSize)+1-min;
            min = Math.min((int)((myHeight-myYCoord)/this.cellSize)+1,
                           (int)(yCoord/this.cellSize)+1);
            this.indexLastCell = (int)(myYCoord/this.cellSize) + min;
        } else if (orientation == 270){
            min = Math.min((int)(myYCoord/this.cellSize),
                           (int)((width-xCoord)/this.cellSize));
            this.indexFirstCell = (int)(myYCoord/this.cellSize)+1-min;
            min = Math.min((int)((myHeight-myYCoord)/this.cellSize)+1,
                           (int)(xCoord/this.cellSize)+1);
            this.indexLastCell = (int)(myYCoord/this.cellSize) + min;
            }
}
```

As shown by the code, the calculus of the minimum strictly depends on the respective orientation of the devices.
Same reasoning was done for ***UP*** and ***DOWN*** swipe directions.

The idea behind this calculus is that the indices of the cells that a device has to send are the same of the ones it has to receive, with just one row (or column) of difference, so we can rely just on our indices to set the value cells in our matrix.
Here's a picture to clarify the idea:

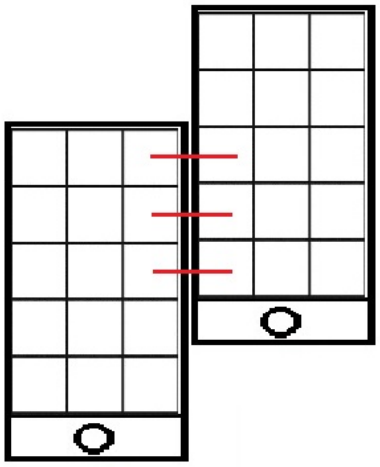

Figure 3.5: Cells correspondence

Each device does not need to calculate the indices of the cells of the other device: it just need to know that it has to send the cells correspondent to the indices calculated, that are the same of the cells that it will receive (considering, of course, that these ones will be set on the "ghost" column or row, since they represent the cells on the other device).

3 - Preparation of cells to be sent

We also have to consider if the list of cells that we will send will have to be reversed or not: the idea is that each device, when receiving the cells to be set, simply scans the list received and sets the values on the correspondent indices, so is responsibility of the sending device to know whether the list it

will send will have to be reversed or not, and this depends (once again) on the orientation of the devices.

```
if(myDir.equals(PinchInfo.Direction.RIGHT) ||
        myDir.equals(PinchInfo.Direction.LEFT)){

    if(orientation == 0){
        this.reverseList = false;
    } else if (orientation == 90){
        this.reverseList = true;
    } else if (orientation == 180){
        this.reverseList = true;
    } else if (orientation == 270){
        this.reverseList = false;
    }
}
```

Once set all of these variables, the device is ready to send its cells: knowing the indices and the direction of its swipe (this is useful to know whether to consider the first or last column - or row - to send to the other device), it fills a list with the values of the cells, and then send it to the paired device. Here's a piece of code to clarify this point:

```
switch(myDir){
    case RIGHT:
        for(int i = this.indexFirstCell; i<=this.indexLastCell; i++){
            cellsToSend.add(matrix[i][columns]);
        };
    break;
```

Since we know that the direction of our swipe is ***RIGHT***, we maintain the value of the column fixed (that is the last one - ghost one) and we scan the

rows from the first index calculated to the last one.

The receiving of the cells follows the same idea: knowing the direction of the device's swipe and the indices correspondent to the cells to set, we fill the matrix with the values received, but this time filling the additional row (or column), not the one showed on the screen.

3.5 Testing

Testing has been a fundamental part of our project. We organized this phase in the following steps:

- Unit testing: in this part we focused our attention on the correct behaviour of each component in our system.
- Integration testing: in this step we verified the correct behaviour and interaction between all the components in the system.

These tests also attempt to verify some properties that the system should own, like deadlock absence.
We also performed some latency testing: in this part we analysed the network latency of the system, in order to test the system scalability.
In the next subsections we will show that the network cost allows the system to scale horizontally up to the server capacity.
For some tests we used "Espresso" to perform some actions on the physical device.

3.5.1 Unit testing

In this subsection we take a closer look at how each component in the system has been tested to ensure its correct behaviour. We split component testing into different classes, and each class has a set of test methods.

TestConnectedDeviceInfo: in this class we test device connection, as well as the informations they should send to each other.

- TestGetter: this test ensures that informations about a device connection are correctly stored (correct names for queues and correct stored direction of swipe).
- CalculateIndex: this test ensures that, given a couple of devices correctly connected, the informations about which cells should be sent and the position of the other device are correctly calculated.

TestGridView: in this class we test the correct behaviour of the GridView visualization, including the swipe detection and the "game of life" calculus.

- TestSwipe: this test ensures that a swipe is correctly detected, differentiating from a straight one (correct) and an oblique one(wrong).
- TestSettingCell: this test ensures that cells are properly setted as alive or dead upon a screen pressure on a grid's square.
- TestGame: this test ensures that the game is correctly started/stopped upon double tap on the screen.

RabbitMQTest: in this class we test the correct behaviour of our server, including queues managing and message forwarding.

- Connect: this test simply ensures the correct connection with the RabbitMQ server.
- TestQueue: this test ensures that queues are correctly added and managed by the server side.
- TestPublishSubscribe: this test ensures that, when a subscriber subscribes to a topic, the subscribe queue and the exchange are correctly created and all messages belonging to that topic are correctly sent.

- TestPublishSubscribe2: this test is the same as the previous one, with the difference that the subscriber is the same as the publisher; this is to verify if the broadcast message is sent and received correctly from the same device, and so it has to be filtered.

3.5.2 Integration testing

In this subsection we finally test the behaviour of the system as a whole.

TestHandler: this class contains all the integration tests.

- TestPinchAndDetachment: this test ensures that, upon the occurrence of a swipe, a couple of devices is correctly connected. After that, we consider the case of the devices' unpairing and test its correct behaviour.

- TestCommunicationBetweenDevice: in this test we take into account a full communication between two devices. In particular, after two devices correctly connect to each other, we test the correct cell sending, as well as the generation calculus and the pause.

- TestCommunicationBetweenDevice2: this test is pretty much the same as the previous one, with the difference that now we test the behaviour of a system composed by three different devices.

- TestCloseCommunication: in this test we consider the case of a closed connection. In particular, we verify the correct closing of a connection between a couple of devices after the sending of the "close connection" message.

- TestCloseCommunicationAfterSendCells: this test insures the correct behaviour of the system in case a device disconnects after sending the cells to its neighbours.

3.5.3 Latency testing

We also did some tests on latency, to check what is the behaviour of the network with multiple devices connected and communicating between them. We were able to perform the tests on four devices, checking how the system scales.
Here the results we collected:

	2 Devices	**3 Devices**	**4 Devices**
Time between generations	50-100 ms	50-100 ms	50-200 ms
Time between swipe messages	35 ms	20 ms	22 ms

Time between generations is referred to the time elapsed between the calculus of one generation and the next one.
Time between swipe messages is referred to the time elapsed between the moment of the swipe on a device and the swipe message that it receives, correspondent to the device which it is trying to pair with.
As we can see from the result, the system scale well with a growing number of devices.

3.6 Teamwork

Before we started any implementation, we concentrated on choosing which solutions were the best for all aspects of our project, starting from the technology beneath the informations exchange up to how we wanted to represent the grid, the cells and what was the best method to send, receive and set the cells of devices pinched together.
After setting these things straight, we elaborated a primary structure of how our program should be like, outlining which the main nuclei of our application would be and how things should be done from a logical point of view (that means, what the dynamic of the distributed communication should be,

how it has to be carried on, how we would face the known problems of a distributed environment).
Having agreed on all these things and having defined the structure and modus operandi of our program, we could start and implement things. Francesco Cozzolino created the utility class for the RabbitMQ methods that we used for communication, and took care of the synchronization between generations; Luca Longobardi and Marina Londei took care of the calculus of correspondent cells when two devices pinched together and the management of the sending - receiving of cells on the queues.
The class containing the informations about the pinching between two devices (class *PinchInfo*) was created by Luca Longobardi, together with the management of the closure of channels when an application is closed.
The management of the swipe gesture and the start and stop of the system with the resulting need of communication was done by Marina Londei.

For the test classes, the work has been divided this way: Marina Londei wrote the tests for the class *TestConnectedDeviceInfo* to check if the informations exchanged between two devices about their cells are correct, the name of the queues and in general the informations about the communication of the couple is correct.
Luca Longobardi wrote the test for the correct behaviour of the GridView and the RabbitMQ methods; Francesco Cozzolino wrote the tests for the *Handler*, checking if the attachment and detachment of devices and the correct communication.

Conclusions

The results we obtained were satisfying, both for the responsiveness of the application and the graphical outcome.
This was one of our first real distributed application, so we had to face all the complexities that this kind of computation gives. The idea of connecting two or more devices by the gesture of *swiping* them together came up, as already said, after reading the paper by Ohta and Tanaka, and we decided to use this idea to reflect the connection of screens that share some kind of content together, and modify their nature as time passes. This vision can be applied to a lot of different areas of interest.
As for RabbitMQ, it keeps the old messages and send them to the receiver as soon as it is available (to realize the *loose coupling we already talked about*; to overcome this behaviour, as we don't want a device to receive messages when it wasn't online (e.g.: when it was stopped), we check the timestamp of messages to see if they were sent before the device was connected; if so, the message is ignored. This prevents any kind of trouble between generations.

As alternative solution, we could have used TuCSoN tuples centre to manage communication. We didn't use this technology because it was not included in the topics of our course, but we consider it could be a valid alternative for future developments.

Bibliography

[1] *Pinch: An Interface That Relates Applications on Multiple Touch-Screen by 'Pinching' Gesture* - Takashi Ohta and Jun Tanaka

[2] *Web of things* - A. Ricci - Course's slides

[3] `https://www.rabbitmq.com`