INFO0010 Introduction to Computer Networking
First part of the assignment

E. Marechal, G. Leduc
Université de Liège

Academic year 2021-2022

Submission before 2nd of November 2021

Abstract

In this assignment, you will have to implement a client-server application using Java Sockets.
The application is a monster-hunting game relying on sensors distributed across the environment, and that are able to detect when a monster is close to them. The sensors share the data they collect by connecting to the main server, which uses the publish/subscribe message pattern. A client can then connect to the server and subscribe to receive the data collected by the sensors. Based on this information, the player will try to deduce the monster's position and make a guess.
Students will work alone for this part of the assignment.

Contents

1 Monster Hunting with Internet of Things (IoT)

The Internet of Things (IoT) describes physical devices, that are equipped with various sensors and technologies, and that exchange data with other devices or systems over the Internet or other communication networks[1].
The set of applications of IoT devices is extremely large, ranging from consumer applications (such as smart watches, smart homes, etc) to medical and healthcare applications. The first-ever connected device dates back to 1982 and was a coca-cola vending machine at the Carnegie Mellon University, which was able to report on its inventory and whether newly loaded drinks were cold or not[1].

In this assignment, we will use the Internet of Things to track the whereabouts of a monster roaming the Grid (a simple 2D environment composed of a 10x10 grid).

The system is composed of:

- A main server, called the *Broker*, that is up for connections from either the IoT sensors or the client. **The *Broker* will be provided to you.**
- A client, called the *Subscriber*, that will interact with the player using terminal I/Os, send messages to the *Broker*, read its responses, and update its display accordingly. **In this project, you are to implement the *Subscriber*.**
- 10x10 IoT sensors placed on every cell of the Grid. The sensors share the data they collect by connecting to the *Broker*. **The sensors will also be provided to you.**

The *Broker* implements a publish/subscribe message pattern, which makes it easy to provide a many-to-one or one-to-many message distribution system. More precisely, every IoT sensor will connect to the *Broker* to publish their data in a particular topic. Topics are a way to structure the Application Messages into different categories (e.g., sensors publish data in the `position` topic).
On the other side, the *Subscriber* will connect to the *Broker* and subscribe to the topics it is interested in, in order to track the monster. Each time the *Broker* receives a message in a particular topic, it looks up if any client has subscribed to this particular topic, and if so, forwards the message to said clients.

1.1 Rules of the Game

The game is played in turns between you and the monster. At each iteration, the monster moves forward one square on the Grid (either up, down, left, right, or diagonally). Once the monster has moved, the sensors measure their environment and then publish the information (in the `position` topic).

[1] https://en.wikipedia.org/wiki/Internet_of_things

Unfortunately, these sensors are not of the best quality and always provide inaccurate measurements. There is also a possibility that some sensors don't detect anything at all, even though the monster is right on their tile. The position they yield is thus associated with a number, expressing the uncertainty in the measurement. This number can either be:

- `1`: The monster is situated at most 1 tile away[2],
- `2`: The monster is situated at most 2 tiles away[2],
- `3`: The monster is situated at most 3 tiles away[2],

As soon as the information is published to the *Broker*, it will send the information back to the *Subscriber* (if it has correctly subscribed to the `position` topic), which will in turn display the information to the player on the terminal. Fig. 1 shows an example of a potential output from the *Subscriber*, interacting with both the player and the *Broker*. Note that the *Subscriber* needs to combine the information sent by the sensors in order to display the zone in which the monster can effectively be (examples of how the information is combined can be found in Appendix B).

Based on this information, the player then has to take a guess on the monster's position (e.g., tile B4). The *Subscriber* then publishes the guess to the *Broker* in the `guess` topic. To know whether the player's guess was right, the *Subscriber* has to subscribe to the `victory` topic. We advise you to make the *Subscriber* subscribe from the start to the `victory` topic, instead of subscribing after submitting the player's guess.

After each turn, the previous sensors' information is discarded, and only the new one must be considered for the player to make their guess.

1.2 Launching the system

The *Broker* must listen on port 2xxx – where xxx are the last three digits of your ULiège ID.

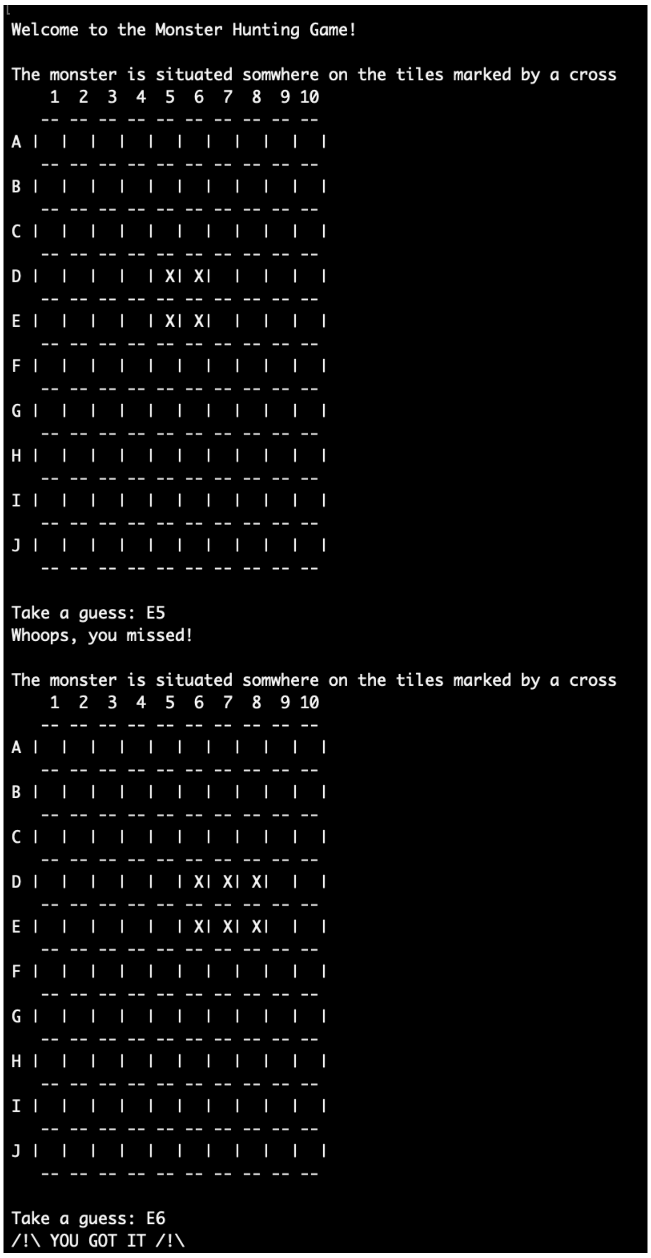
Your software is to be invoked on the command line, with one additional argument for the port number of the server it should connect to:

```
java Subscriber 2xxx
```

The *Broker* as well as the IoT sensors are invoked on the command line like this:

```
java Broker_and_Sensors 2xxx
```

[2]The distance in the Grid is measured in tiles. Two neighboring tiles (i.e., tiles that are in contact, either by a side or by a corner) are considered to be 1 tile away from each other.



```
Welcome to the Monster Hunting Game!

The monster is situated somwhere on the tiles marked by a cross
    1  2  3  4  5  6  7  8  9 10
   -- -- -- -- -- -- -- -- -- --
A |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
B |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
C |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
D |  |  |  |  | X| X|  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
E |  |  |  |  | X| X|  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
F |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
G |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
H |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
I |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
J |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --

Take a guess: E5
Whoops, you missed!

The monster is situated somwhere on the tiles marked by a cross
    1  2  3  4  5  6  7  8  9 10
   -- -- -- -- -- -- -- -- -- --
A |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
B |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
C |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
D |  |  |  |  |  | X| X| X|  |  |
   -- -- -- -- -- -- -- -- -- --
E |  |  |  |  |  | X| X| X|  |  |
   -- -- -- -- -- -- -- -- -- --
F |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
G |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
H |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
I |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --
J |  |  |  |  |  |  |  |  |  |  |
   -- -- -- -- -- -- -- -- -- --

Take a guess: E6
/!\ YOU GOT IT /!\
```

Figure 1: Interaction between the *Subscriber* and the player

1.3 Cheating Mode

In order to make development and testing easier for you, the *Broker* and the IoT devices can be launched in cheating mode by typing:
`java Broker_and_Sensors 2xxx cheat`

The cheating mode allows you to know at each iteration the path taken by the monster on the Grid, as well as the messages you will receive from the sensors, in order to verify that your implementation is consistent. This information is displayed on the terminal by the `Broker_and_Sensors` program.

2 Communication with the Broker: A custom-made binary protocol

The Monster Hunting Protocol (MHP) is a custom-made protocol designed for this assignment only. This protocol is binary-oriented, meaning that all exchanges are custom packed and not intended to be human-readable (as opposed to "text-oriented" protocols such as HTTP that is based on readable lines of text).

Each message is called a Control Message and is made of a header of 3 bytes, possibly followed by more data, called the payload. Examples of MHP Control messages can be found in Appendix A

2.1 Header

- The first byte is the Version Number, allowing several versions of the protocol to be supported. In this assignment, the protocol version is 1.
- The second byte identifies the Control Message Type. The possible values and their meaning are explained in Sec. 2.3.
- The third byte identifies the length of the payload in bytes. Note that you can only encode a payload of maximum 255 bytes, as there is only one byte to encode the length.

2.2 Payload

The payload of an MHP Control message will always contain text, which will be encoded as UTF-8 strings. Several strings can be present in the payload. In order to delimit the boundaries between the strings, each string will be prefixed with one byte (giving the length of the current string in bytes) allowing so to parse the payload. The content of the payload will vary based on the MHP Control Message Type.

Examples of payload encoding can be found in Appendix A.

2.3 MHP Control Messages

In this section, we review the different types of Control Messages that can be exchanged between the *Broker* and the *Subscriber*.

- SUBSCRIBE: Control Message Type 0
  This message allows the *Subscriber* to subscribe to a certain topic it is interested in, and can only be sent from the client to the server.

  The payload of this message will contain two utf-8 strings, the first one for the *Subscriber*'s identifier, and the second one for the topic it is interested in.

- PUBLISH: Control Message Type 1
  This message allows the *Subscriber* or the *Broker* to publish information to the other party, and can thus be sent from the server to the client, or from the client to the server.

  The payload of this message will contain two utf-8 strings, the first one for the topic the Application Message is being published to, the second one for the Application Message itself.
  The Application Message refers to the useful information that is used to actually play the game (such as the sensor data, or the player's guess). The Application Messages format is explained in Sec. 2.4

- ACK: Control Message Type 2
  An ACK message can either be sent from the server to the client, or from the client to the server, and has to be sent right after a SUBSCRIBE or a PUBLISH message has been received, in order to acknowledge the message, or potentially return an error.

  The payload of this message will contain one or two utf-8 strings, depending on whether the message was processed without problem, or if there was an error. In case of error, the first utf-8 string will contain the word `ERROR`, and the second utf-8 string will contain more information on what happened (e.g., `Unknown Control Message Type`). In case everything went fine, there will only be one utf-8 string containing the word `OK`.

2.4 MHP Application Messages

The Application Messages (as part of the payload of PUBLISH Control Messages) can contain three different types of information, based on the topic they have been published to:

- Sensor data: Published in the `position` topic, this data will be formatted as a utf-8 string by stating the sensor's position on the Grid, followed by the measure it made, separated by a semi colon: `"[tile]:[measure]"`.
  `[tile]` will be a correct position on the Grid (e.g., `B4`, `D5`, etc), and `[measure]` will either be `1`, `2`, or `3` (corresponding to the different levels of confidence of the sensor's measurement - see Sec. 1.1).

- Guess made by the player: Published in the `guess` topic, this data will be formatted as a utf-8 string, simply stating the player's guess as: `"[tile]"`.

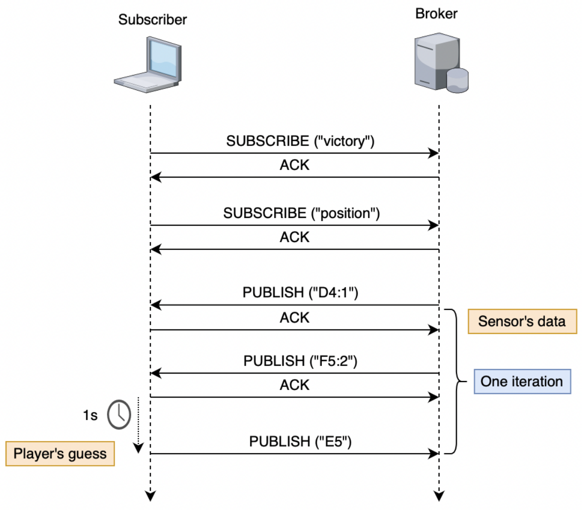


Figure 2: MHP Operational Mode. After having subscribed to the `victory` and `position` topics, the *Subscriber* waits for the sensors' measurements to be delivered. After 1s of receiving nothing from the *Broker*, the `Subscriber` will stop listening and process the sensors' data.

- Victory announcement: Published in the `victory` topic, this data will be formatted as a utf-8 string, simply stating : `"victory!"`.

2.5 Incomplete or incorrect commands

When dealing with network connections, you can never assume that the other side will behave as you expect. It is thus your responsibility to check the validity of the messages you receive (and respond with the appropriate message if something goes wrong). In this project, if the message received from the *Broker* contains errors or is not well formatted, the *Subscriber* is expected to send an Ack Control Message with an error and to close the connection right after.

2.6 MHP operational mode

With the publish/subscribe pattern, it is impossible to actually know how many messages will be received by your *Subscriber* from the sensors at a given iteration. Indeed, after having subscribed to the `position` topic, or after having submitted a guess to the `guess` topic, the `Broker` is susceptible to send multiple messages from the sensors to you, as part of the current iteration. A visual summary of the MHP operational mode can be found in Fig. 2.

Therefore, we will consider it safe for the `Subscriber` to stop listening for new messages after a certain amount of time receiving nothing on the wire. This waiting time will be set to 1s, after which it is very unlikely to receive anything more from the

Broker. You can set a timeout on a socket with `socket.setSoTimeout(1000)`. After receiving nothing for 1s, the `read` operation will trigger a `SocketTimeoutException`.

3 TCP: A stream-oriented protocol

The MHP relies on the TCP protocol, which is stream-oriented. This has at least three implications:

1. You can be sure that the connection is lossless and that there won't be any packet re-ordering.
2. The sender could be buffering the output, i.e. the data that is requested to be sent might not have been effectively sent yet. This can be solved by calling the flush() method on the OutputStream. Nagle's algorithm can also prevent a packet from departing immediately. It can be deactivated using setTcpNoDelay(true).
3. **The data requested to be sent with one call to a write method will not necessarily be received with only one read method from the other side. Several reads could be necessary to recover the entire information.** This is because we are reading from a stream, not messages (as we would with UDP, for instance). We thus have to find a way to delimit message boundaries on the stream.

   MHP achieves this by having a constant-size header, whose "remaining length" field determines how many bytes, if any, will follow. In practice, you should thus come up with a way to recover a message-oriented communication scheme using a stream protocol such as TCP. As a side note, it is **not good practice** to read byte by byte from the stream.

4 Guidelines

- You will implement the programs using Java 1.8, with packages `java.lang`, `java.io`, `java.net` and `java.util`,
- You will ensure that your program can be terminated at any time simply using CTRL+C, and avoid the use of ShutdownHooks
- You will not manipulate any file on the local file system.
- You will ensure your main class is named Subscriber, located in Subscriber.java at the root of the archive, and does not contain any `package` instruction.
- You will not cluster your program in packages or directories. All java files should be found in the same directory.
- You will ensure that your program is fully operational (i.e. compilation and execution) on the student's machines in the lab (ms8xx.montefiore.ulg.ac.be).

Submissions that do not observe these guidelines could be ignored during evaluation.

Your commented source code will be delivered no later than 2nd of November 2021 (**Hard deadline**) to the Montefiore Submission platform (http://submit.run.montefiore.ulg.ac.be/) as a .zip package.

Your program will be completed with a report called report.pdf (in the zip package) addressing the following points:

Software architecture: How have you broken down the problem to come to the solution? Name the major classes and methods responsible for requests processing.

Message-oriented communication using a stream: Explain how you handled the recovery of a message-oriented communication scheme using a stream protocol such as TCP.

Limits & Possible Improvements: Describe the limits of your program, especially in terms of robustness, and what you could have done better.

A MHP Control Message examples

In this section, you can find various examples of MHP Control Messages in Tables 1, 2, and 3.

	7	6	5	4	3	2	1	0	
byte 0	0	0	0	0	0	0	0	1	Version Number
byte 1	0	0	0	0	0	0	0	0	Message Type
byte 2	0	0	0	0	0	1	1	1	Remaining length
byte 3	0	0	0	0	0	0	1	1	utf-8 byte length (first string)
byte 4	0	1	1	1	0	1	0	0	t
byte 5	0	1	1	0	1	0	0	1	i
byte 6	0	1	1	0	1	1	0	1	m
byte 7	0	0	0	0	0	0	1	0	utf-8 byte length (second string)
byte 8	0	1	1	0	0	0	0	1	a
byte 9	0	1	1	0	0	0	1	0	b

Table 1: SUBSCRIBE message of a client with identifier `tim` who wants to subscribe to the `ab` topic

	7	6	5	4	3	2	1	0	
byte 0	0	0	0	0	0	0	0	1	Version Number
byte 1	0	0	0	0	0	0	0	1	Message Type
byte 2	0	0	0	0	1	0	0	0	Remaining length
byte 3	0	0	0	0	0	0	1	0	utf-8 byte length (first string)
byte 4	0	1	1	0	0	0	0	1	a
byte 5	0	1	1	0	0	0	1	0	b
byte 6	0	0	0	0	0	1	0	0	utf-8 byte length (second string)
byte 7	0	1	0	0	0	0	1	0	B
byte 8	0	0	1	1	0	1	0	0	4
byte 9	0	0	1	1	1	0	1	0	:
byte 10	0	0	1	1	0	0	1	1	3

Table 2: PUBLISH message in the `ab` topic, with Application Message data (sensor data)

	7	6	5	4	3	2	1	0	
byte 0	0	0	0	0	0	0	0	1	Version Number
byte 1	0	0	0	0	0	0	0	1	Message Type
byte 2	0	0	0	0	1	0	0	1	Remaining length
byte 3	0	0	0	0	0	1	0	1	utf-8 byte length (first string)
byte 4	0	1	1	0	0	1	1	1	g
byte 5	0	1	1	1	0	1	0	1	u
byte 6	0	1	1	0	0	1	0	1	e
byte 7	0	1	1	1	0	0	1	1	s
byte 8	0	1	1	1	0	0	1	1	s
byte 9	0	0	0	0	0	0	1	0	utf-8 byte length (second string)
byte 10	0	1	0	0	0	1	0	0	D
byte 11	0	0	1	1	0	1	0	1	5

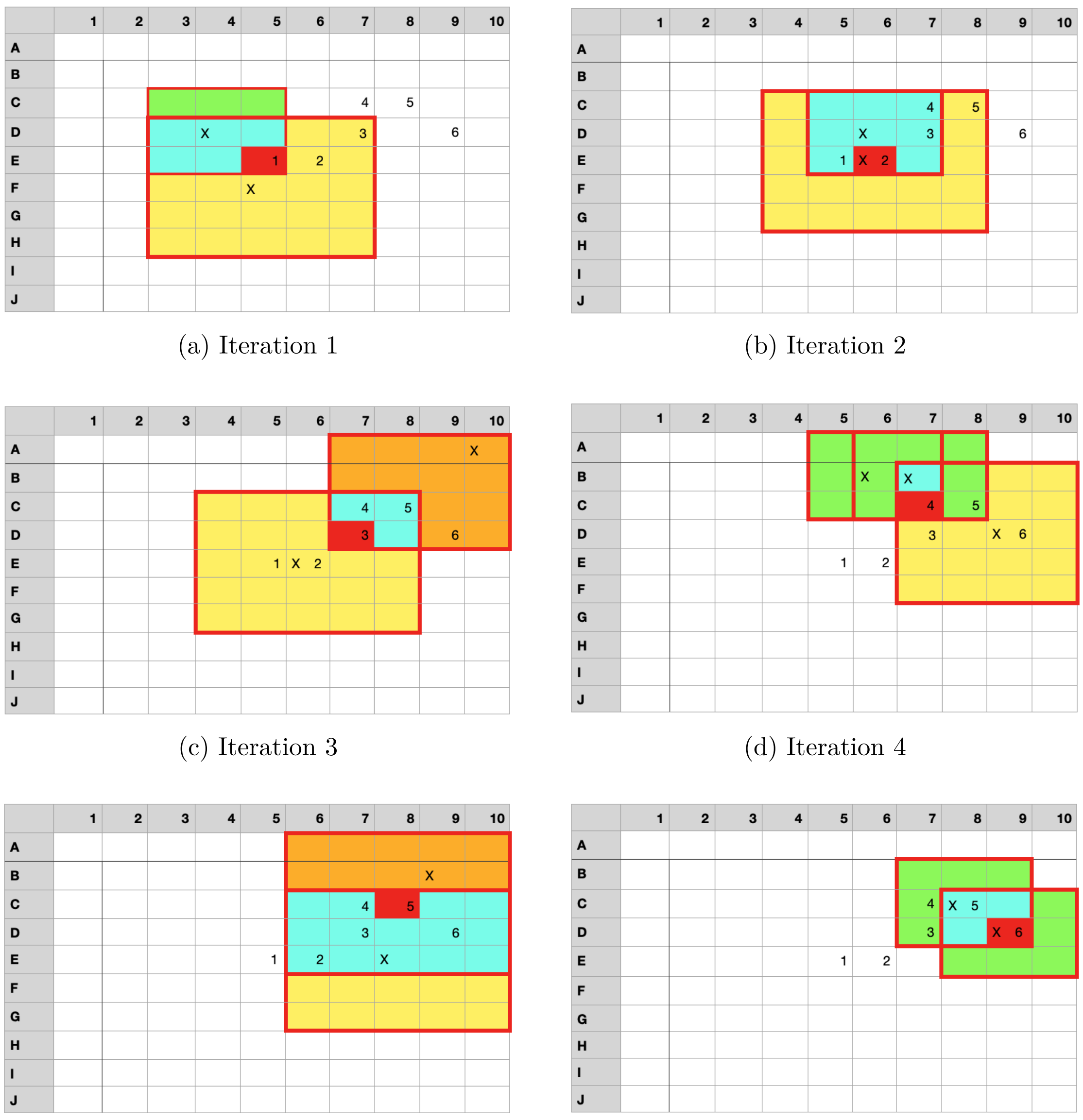
Table 3: PUBLISH message in the `guess` topic, with Application Message data (player's guess)

B A Game of Monster Hunting

In this section, you are presented with an example of a game of Monster Hunting. This example also shows you how the various information from the sensors have to be combined in order to display the effective zone in which the monster can be situated.

Across all iterations, the path taken by the monster is showed with successive numbers from 1 to 6, indicating on which tile the monster will be at the corresponding iteration.
At each iteration, the current position of the monster is displayed in red. The sensors that were activated for the iteration are represented by a cross, and the zone they cover is showed with a NxN color square around the sensor. The confidence of the sensor (i.e., whether the monster is situated at most 1, 2, or 3 tiles away from the sensor) is showed with different colors: green for a 3x3 square, yellow for a 5x5 square, and orange for a 7x7 square. After combining all sensors' information, the effective zone where the monster can be situated is shown in blue.

For example, at iteration 1, the monster is situated on tile `E4`, and the sensors on tiles `D4` and `F5` respectively fired with values 1 and 2. If we combine both sensors' information, this means the monster is situated on the Grid somewhere in the square from `D3` to `E5`.



(a) Iteration 1

(b) Iteration 2

(c) Iteration 3

(d) Iteration 4

(e) Iteration 5

(f) Iteration 6

Figure 3: Monster Hunting Game Example