

Building Evacuation System

José Miguel Ferreira Simões

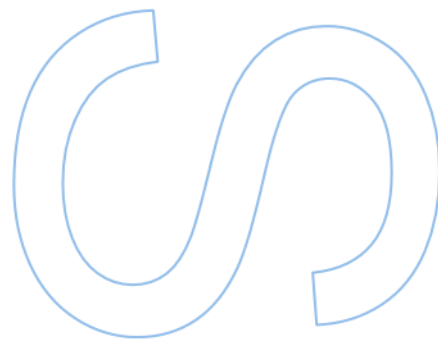
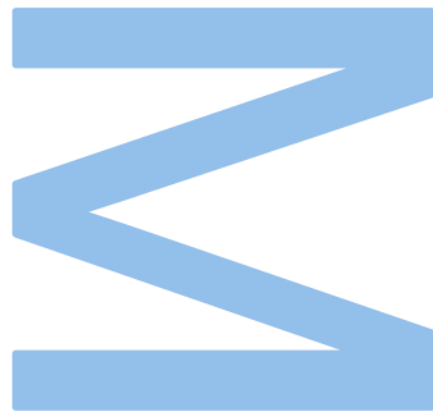
Mestrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciências de Computadores
2023

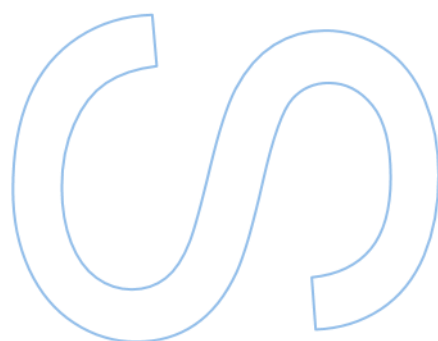
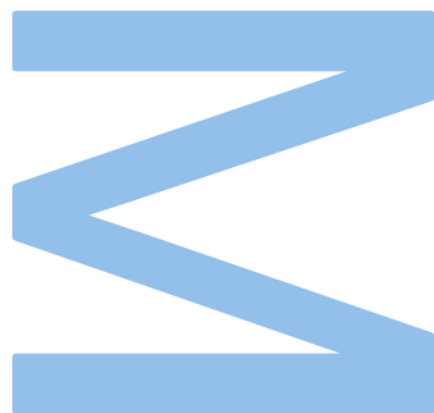
Orientador

Rui Prior, Professor, Faculdade de Ciências da Universidade do Porto

Coorientador

Sérgio Crisóstomo, Professor, Faculdade de Ciências da Universidade do Porto





Resumo

Atualmente vemos a construção de edifícios cada vez mais complexos, com os centros urbanos a terem um crescente número de edifícios de grandes dimensões, nos quais se incluem, por exemplo, arranha céus ou shoppings.

Uma das principais características destes edifícios que não conseguiu acompanhar a sua evolução arquitetónica foram os planos de evacuação em caso de emergência, com a evacuação na maioria dos edifícios a continuar baseada em planos estáticos. Estes planos podem fornecer informações erradas e que colocam as pessoas em perigo, visto que as guiam sempre numa direção predefinida, não tendo em conta a localização e características específicas de cada emergência.

O objetivo deste projeto é desenvolver um sistema dinâmico que permita o cálculo do caminho mais seguro para evacuar edifícios durante emergências. Neste relatório é proposta uma solução, baseada numa arquitetura distribuída, com o edifício dividido em diferentes áreas, e composta por diferentes componentes. Uma unidade central capaz de receber e processar informações acerca da planta de um edifício, utilizando um formato padrão de mapeamento interno. Cada área do edifício irá conter um ou mais nós decisores, capazes de monitorizar aquela zona, com ajuda da informação recebida por sensores, e com essa informação calcular a rota de evacuação mais segura. Estes nós decisores em diferentes áreas terão ainda capacidade de comunicar entre si, transmitindo informação resumida, de forma a todos poderem ter uma visão geral do edifício. Depois de calculada a rota mais segura, esta informação é transmitida aos utilizadores, através de dispositivos presentes no edifício, de forma a que estes evitem zonas afetadas pela ocorrência e possam sair do edifício da forma mais segura possível.

Abstract

Nowadays we see the construction of increasingly complex buildings, with urban centers having an increasing number of large buildings, including skyscrapers or shopping malls.

One of the main features of these buildings that failed to keep up with their architectural evolution was the evacuation plans in case of emergency, with evacuation in most buildings continuing to be based on static plans. These plans can provide incorrect information that puts people in danger, as they always guide them in a predefined direction, without considering the location and specific characteristics of each emergency.

The objective of this project is to develop a dynamic system that allows the calculation of the safest path to evacuate buildings during emergencies. In this report, it is proposed a solution, based on a distributed architecture, with the building divided into different areas, and composed by different components. A central unit capable of receiving and processing information about the building's plan, using a standard internal mapping format. Each building area will contain one or more decision nodes, capable of monitoring that area, with the help of information received by sensors, and using this information to calculate the safest evacuation route. These decision nodes in different areas will also have the ability to communicate with each other, transmitting summarized information, so that everyone can have an overview of the building. After calculating the safest route, this information is transmitted to users, through devices present in the building, so that they avoid areas affected by the incident and can leave the building in the safest way possible.

Acknowledgements

First of all, I would like to thank all the people who directly and indirectly contributed to the completion of this project and who always supported me, throughout this academic journey, especially in the most difficult moments.

More particularly, I would like to thank my family for their unconditional support, especially my parents, my sister and my grandparents who have always been by my side, encouraging me to complete this journey with success and dedication.

To my group of friends and classmates who were fundamental support!

To all the teachers who left a mark on the way I see this giant world of computers.

And finally, a thank you to professors Rui Prior and Sérgio Crisóstomo, for the opportunity to work with them on this project, helping me with every difficulty and always being available for everything.

I would also like to thank Institute of Telecommunications (UIDB/50008/2020) for hosting this work.

To all of you, thank you very much!

Contents

Resumo	i
Abstract	iii
Acknowledgements	v
Contents	xi
List of Tables	xiii
List of Figures	xvii
Listings	xix
Acronyms	xxi
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	3
1.3 Dissertation structure	4
2 Background	7
2.1 Building Representation	7
2.1.1 IMDF	7
2.1.2 IndoorGML	9
2.2 System Architecture	11

2.2.1	Centralized Systems	12
2.2.2	Decentralized Systems	12
2.2.3	Distributed Systems	13
2.3	Search Algorithms	13
2.3.1	Dijkstra’s algorithm	13
2.3.2	Floyd-Warshall algorithm	14
2.3.3	Bellman-Ford algorithm	15
2.4	Databases	15
2.4.1	Relational Databases	15
2.4.2	Non-Relational Database	17
2.5	Network Simulation and Emulation Tools	18
2.5.1	GNS3	18
2.5.2	Mininet	19
2.5.3	EVE-NG	20
2.5.4	NS-3	21
2.5.5	Netem	22
3	State of the Art	25
3.1	A Distributed Decision Support System	25
3.1.1	Assumptions	25
3.1.2	Graph Representation of the Building	26
3.1.3	System Architecture	27
3.1.4	Safest Path Result	27
3.1.5	Conclusion	28
3.2	A Smart Evacuation Guidance System for Large Buildings	28
3.2.1	System Architecture	29
3.2.2	Smart Indicators Design	30
3.2.3	LCDT-Based Weighted Graph for Finding Evacuation Routes	30

3.2.4	Conclusion	31
3.3	Emergency Evacuation using Wireless Sensor Networks	31
3.3.1	Building Representation	32
3.3.2	Hazard Time Computation	33
3.3.3	Safest path calculation	33
3.3.4	Results	34
3.4	IoT-Based Intelligent Modeling of Smart Home Environment for Fire Prevention and Safety	35
3.4.1	ZigBee Communication Protocol	35
3.4.2	System Overview	36
3.4.3	Algorithm	37
3.4.4	Conclusion	37
3.5	Summary	38
4	System Design	41
4.1	System Overview	41
4.2	System Architecture	42
4.2.1	Building Representation	44
4.2.2	Configuration Server	46
4.2.3	Decision Nodes	47
4.2.4	Sensor Nodes	48
4.2.5	Guidance Nodes	48
4.3	Navigation Graph	49
4.4	Navigation Algorithm	51
4.5	Communication Algorithm Between Decision Nodes	53
4.5.1	Loops Problem in Navigation Graph	56
4.6	Location of Sensor Nodes	58
4.7	Location of Guidance Nodes	58

4.8	Model and Communication Protocols Between Components	59
4.8.1	Decision Node $\leftrightarrow$ Decision Node and Decision Node $\leftrightarrow$ Guidance Nodes .	59
4.8.2	Sensor Node $\rightarrow$ Decision Node	60
4.9	Database	61
5	System Implementation	65
5.1	System Programming Language	65
5.2	Server	66
5.2.1	IndoorGML File Processing	66
5.2.2	Connection with database	68
5.3	Sensor Node	69
5.4	Decision Node	71
5.4.1	Navigation Graph Construction	72
5.4.2	Shortest Path Algorithm	74
5.4.3	Communication Algorithm between Decision Nodes	75
5.4.4	Loops Problem in Navigation Graph	77
5.4.5	Communication between system components	78
5.4.6	Evacuation Process and Communication with Guidance Nodes	81
5.5	Guidance Node	83
6	System Evaluation	87
6.1	DCC Building Deployment	87
6.2	Test Conditions	89
6.3	Test Cases	90
6.4	Results	92
6.4.1	Test Case 1	92
6.4.2	Test Case 2	94
6.4.3	Test Case 3	98
6.4.4	Test Case 4	101

6.5	Results Overview	104
7	Conclusion	105
7.1	Future Work	106
7.1.1	Congestion of evacuation paths	107
7.1.2	How to recover from system reboots	107
7.1.3	Security Mechanisms	107
7.1.4	Guidance Nodes Interface	107
7.1.5	Duplicated Decision Node	107
	Bibliography	109

List of Tables

4.1	Temperature intervals and correspondent Danger Levels	48
4.2	Gas/Smoke intervals and correspondent Danger Levels	48
4.3	Building table fields and descriptions	62
4.4	Area table fields and descriptions	62
4.5	Node table fields and descriptions	63
4.6	Transition table fields and descriptions	63
6.1	Bandwidth, Delay and Packet Loss values for each Network Condition	89

List of Figures

1.1	Number of skyscrapers in 21st Century	2
1.2	Traditional Building Emergency Plan	3
2.1	ArcGIS Indoor Routing, retrieved from [10]	9
2.2	InEditor Software	11
2.3	System Architectures Overview	12
2.4	Example relational database structure, retrieved from [13]	16
2.5	Types of NoSQL Databases, retrieved from [12]	17
2.6	GNS3 Graphical User Interface, retrieved from [18]	19
2.7	MiniEdit GUI	20
2.8	NS-3 Network Topology Diagram Tool, retrieved from [18]	22
3.1	Graph of building representation	26
3.2	Example of sensor and link between two decision nodes	26
3.3	Sensors and Decision Nodes deployed in the building	27
3.4	Example safest path calculation	28
3.5	Overview of System Architecture	30
3.6	Smart Indicator Components	30
3.7	Hazard graph	32
3.8	Navigation graph	32
3.9	Calculated hazard times [3]	33
3.10	Example safety path	35

3.11 Overview of System Architecture	36
3.12 System Decision Diagram	37
4.1 Building division by areas	43
4.2 Proposed System Architecture	44
4.3 Example of IndoorGML Navigation Paths	45
4.4 Example of sensor layer in IndoorGML	45
4.5 Arrays of distances by danger level	50
4.6 Example of Navigation Graph with Neighboring Areas	51
4.7 Example of Navigation Graph	53
4.8 Example of Navigation Graph with Dummy Node	53
4.9 Initial Navigation Graph	54
4.10 Navigation Graph before data exchange between Decision Nodes	55
4.11 Navigation Graph Update	56
4.12 Example of loop in navigation graph	57
4.13 Example of Sensor Nodes Location	58
4.14 ER Diagram	61
5.1 Sensor Node Console Application	70
5.2 Navigation Graph Image	83
5.3 Guidance Node Console Application	84
6.1 DCC Floor 0	87
6.2 DCC Floor 1	87
6.3 DCC Floor 0 Navigation Paths	88
6.4 DCC Floor 1 Navigation Paths	88
6.5 DCC Floor 2 Navigation Paths	89
6.6 DCC Floor 0 Default Exit Paths	91
6.7 DCC Floor 1 Default Exit Paths	91

6.8	DCC Floor 2 Default Exit Paths	92
6.9	Communication on a normal network (left) and on a worst network (right)	93
6.10	Test Case 1: DCC Floor 0 Exit Paths	93
6.11	Test Case 1: DCC Floor 1 Exit Paths	94
6.12	Test Case 1: DCC Floor 2 Exit Paths	94
6.13	Test Case 2: DCC Floor 0 Exit Paths	95
6.14	Test Case 2: DCC Floor 1 Exit Paths	95
6.15	Test Case 2: DCC Floor 2 Exit Paths	96
6.16	Test Case 2: Time to deliver the last message with 10 seconds between message resend	97
6.17	Test Case 2: Time to deliver the last message with 5 seconds between message resend	98
6.18	Test Case 3: DCC Floor 0 Exit Paths	99
6.19	Test Case 3: DCC Floor 1 Exit Paths	99
6.20	Test Case 3: DCC Floor 2 Exit Paths	100
6.21	Next Node to follow by users in Node S27	100
6.22	Test Case 3: Message retransmission	101
6.23	Test Case 4: DCC Floor 0 Exit Paths	102
6.24	Test Case 4: DCC Floor 1 Exit Paths	102
6.25	Test Case 4: DCC Floor 2 Exit Paths	103
6.26	Next Node to follow by users in Node S27 and Node S51	103

Listings

4.1	Json structure for Sensor Message	60
5.1	Read GML Nodes	67
5.2	Discover Transition Area	68
5.3	Call Stored Procedure to Get Transitions	69
5.4	Call Stored Procedure to Get Nodes	69
5.5	Sensor Node	70
5.6	Navigation Graph Construction	72
5.7	Dijkstra Algorithm Result	75
5.8	Decision Nodes Communication Algorithm	75
5.9	Choose Best Exit	77
5.10	Update Edge Weight	79
5.11	Send Confirmation Message	81
5.12	Next Guidance Node To Follow	82
5.13	Guidance Node ConsoleApp	84

Acronyms

API	Application Programming Interface	IMDF	Indoor Mapping Data Format
BSON	Binary JSON	JSON	JavaScript Object Notation
CLI	Command-Line Interface	NoSQL	Not Only SQL
CNN	Convolutional Neural Network	OGC	Open Geospatial Consortium
DB	Database	SMS	Short Message Service
DBMS	Database Management System	SQL	Structured Query Language
DCC	Departamento de Ciência de Computadores	TCP	Transmission Control Protocol
FCUP	Faculdade de Ciências da Universidade do Porto	UDP	User Datagram Protocol
GML	Geographic Markup Language	USB	Universal Serial Bus
GUI	Graphical User Interface	XML	Extensible Markup Language

Chapter 1

Introduction

In the last years, we have seen the construction of huge and more and more complex buildings, from giant shopping centers to residential buildings with hundreds of meters of height. These buildings brought comfort to our daily lives and tasks, but they require more complex security measures than small buildings, and there is one aspect that could not keep up with this evolution, the evacuation plans in case of emergency. These types of buildings must be prepared to be frequented by thousands of people daily, which requires a highly efficient method of evacuation in case of any type of accident, from fires to earthquakes and gas leaks, among other hazards, in order to be able to evacuate a large number of people in a reduced time and, consequently, save their lives.

1.1 Motivation

Every day there are a large number of occurrences of accidents, that can be caused by human error but also by natural causes. We cannot control them, but we have the ability to prevent and prepare ourselves for such situations, in order to minimize the damage and lives lost because of it.

In emergency scenarios, the fast evacuation of victims is one of the most important factors to reduce deaths and injuries. Many people die every year in emergency situations due to the absence of efficient evacuation plans, which results in long evacuation times that can put lives at risk.

According to [7], the United Nations estimates that in 2050 two thirds of people will be living in urban areas. With the existing limited space, the tendency is for more and more skyscrapers to be built to accommodate all these people and companies, which is already occurring in the most populated cities in the world.

As we can see in Figure 1.1, the number of skyscrapers higher than 300 meters is growing at a high rate every year, which confirms the tendency mentioned in [7]. These huge buildings

require modern emergency plans that can evacuate a large number of individuals distributed on different floors.

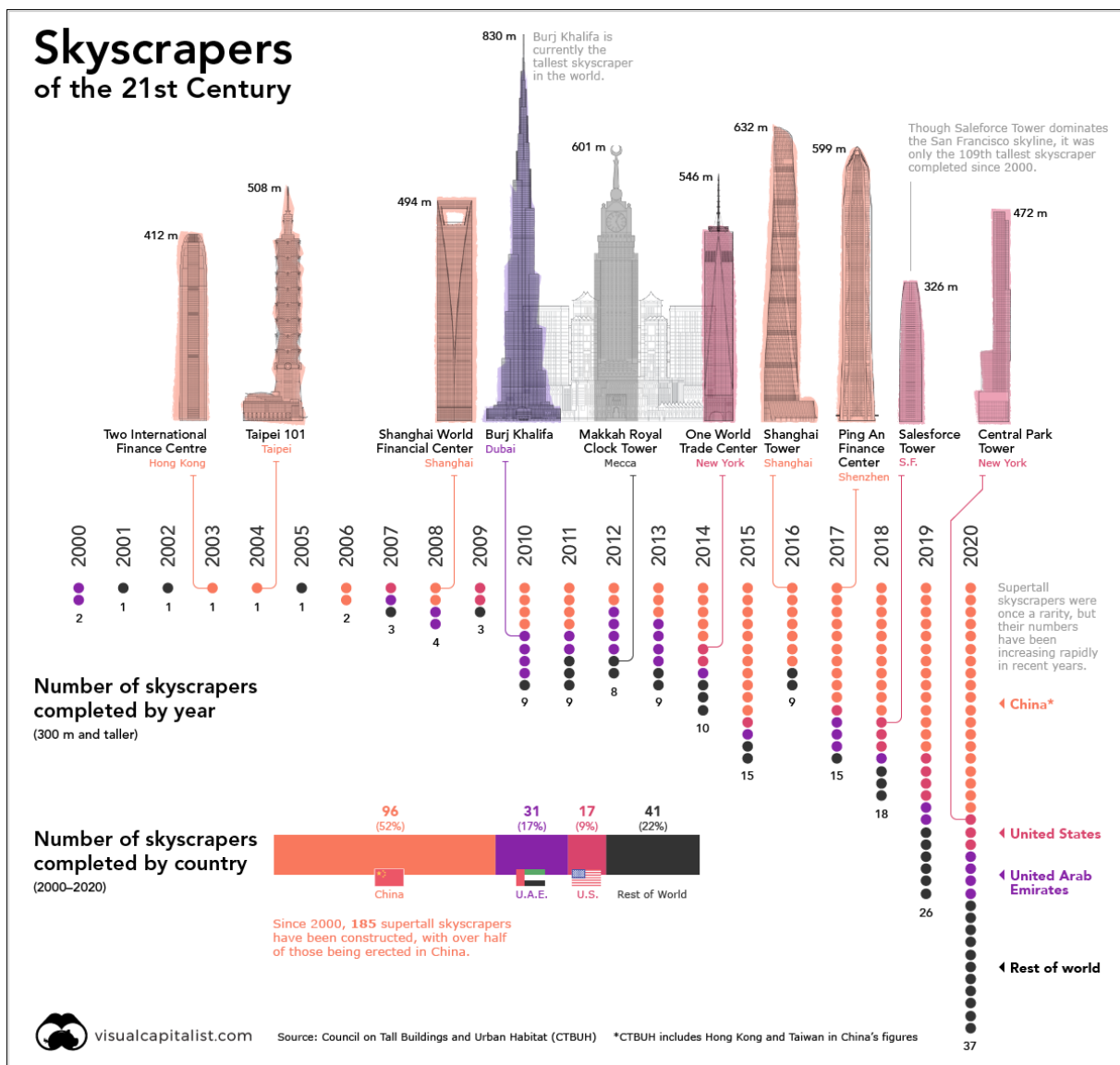


Figure 1.1: Number of skyscrapers in 21st Century

Image source: <https://www.visualcapitalist.com/charting-the-last-20-years-of-supertall-skyscrapers/>

Nowadays, the majority of emergency plans are still static, as exemplified in Figure 1.2, guiding people through a specific exit route, for example, the shortest path to exit the building. The problem with this approach is that this type of plan does not consider the fact that the shortest path can also be affected by the incident or high congestion, making it difficult or even impossible to evacuate people safely. A dynamic approach can represent a huge upgrade to the normal plans because, by collecting the information of various sensors, this model can determine the shortest path to the exit while ensuring that this path is the best choice possible and minimizes the risks that people may run.

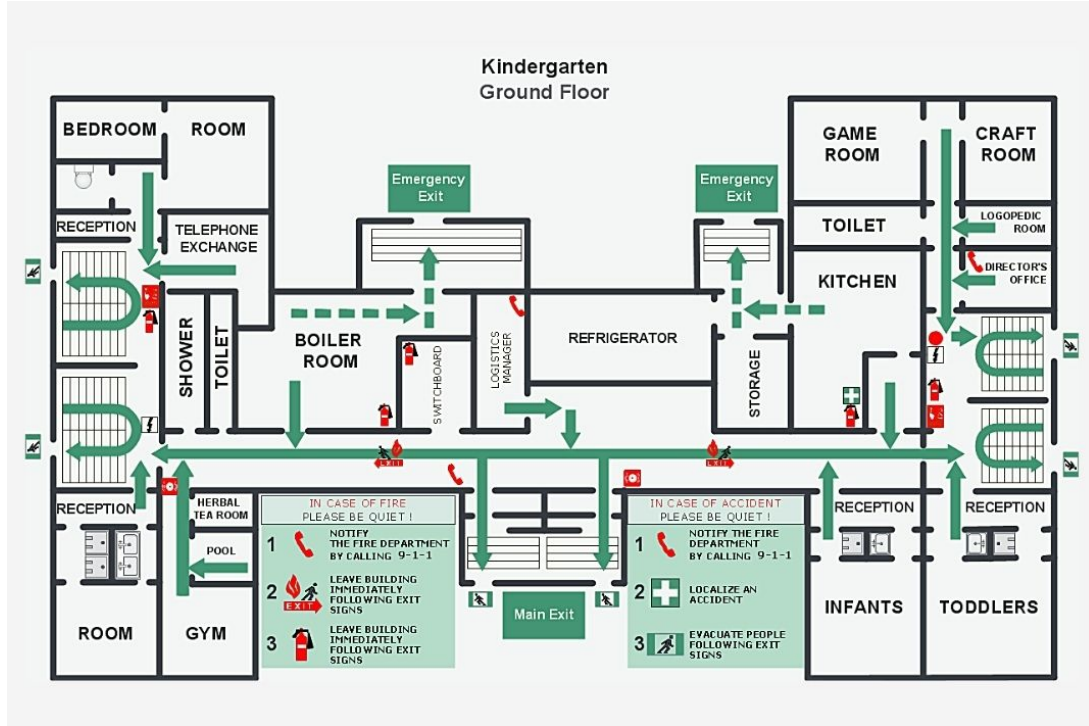


Figure 1.2: Traditional Building Emergency Plan

1.2 Objectives

This work aims to develop a robust intelligent evacuation system, capable of operating in different conditions encountered during emergencies, monitoring the environmental conditions through the use of sensors, calculating the safest path to evacuate the building and indicating that to the users, in order to exit the building safely.

To make this possible, the system needs to know the building's floor plans and its indoor characteristics, in order to be able to create a computational representation of the building. To achieve this goal, a standard mapping format will be used, allowing to mapping indoor different types of indoor spaces, including offices, halls or corridors in the building, which will allow the system to calculate the safest path to an emergency exit.

One of the most important parts of this system is the monitoring of the ambient conditions of the building, through sensors, which will always be alert and inform the system when detecting an abnormal value. The presence of smoke, high temperatures or gas are the main causes of accidents and, as such, will have to be monitored. The unit responsible for reading the ambient values will have to be in constant contact with the safest route calculation system, so that it can update the best route as the ambient conditions vary.

The unit responsible for calculating the safest path, using an algorithm to obtain this path and all the information from the building and the sensors, will be responsible for communicating

this decision to the interface that will indicate to users which direction to follow to leave the building safely.

In emergency environments there are always several adversities and situations such as the lack of electricity, failure in the internet connection or even the case of system components being affected by the accident constitute situations that have to be taken into account in the development of the system, since it will be under these conditions that it will be used and most necessary.

In resume, we want to achieve the following objectives:

- A system capable of importing a standard mapping format.
- Use of sensors to monitor the building conditions.
- A system capable of operating even when some component fail.
- Implementation of mechanisms that guarantee the correct exchange of information between system components and data integrity in adverse conditions.
- A system that can use information collected by different sensors and map this information correctly in order to understand where the emergency is occurring and how much the building is affected.
- With all information collected, the system needs to find the safest path to reach an exit, and transmit this information to a component that allows users to understand which direction they need to follow.
- Study of different interfaces to indicate to users the direction to follow, exploring pros and cons and how they can be included in a system like this.

1.3 Dissertation structure

The structure of this document is the following:

Chapter 1: Introduction: Brief introduction to the subject of emergency evacuation systems, explaining the motivation behind this work and the objectives to achieve in his development.

Chapter 2: Background: In this chapter will be presented important topics that will be mentioned throughout this thesis or are related to the field of evacuation systems.

Chapter 3: State of the Art: Presents the study of state of the art, exploring the different solutions to the indoor evacuation problem. This chapter discusses and explains the various approaches proposed by different authors and their results.

Chapter 4: System Design: This chapter describes the proposed system design, architecture and developed algorithms, specifying in detail the theory of how it works, what innovations are present, and how it should be implemented. This chapter also analyzes the various possibilities and choices made in the design of the system.

Chapter 5: System Implementation: Explain how the system was developed, showing code implementations and specific details present in the tools that were used.

Chapter 6: System Deployment and Evaluation: This chapter analyzes the system performance under different conditions, through the use of emulators, exposing the system to different network problems and finally discussing the results obtained.

Chapter 7: Conclusions and Future Work: Conclusion of the dissertation, presenting some general observations and discussing what objectives were achieved and which may be subject to improvement in future work.

Chapter 2

Background

This chapter will present important topics that will be mentioned throughout this thesis or are related to the field of evacuation systems. Notions and mechanisms that inspired and impacted the development of the project, such as different indoor mapping format standards available for building representation, algorithms for searching shortest paths in graphs, sensor monitoring services, different methods of storing data, tools for analyzing and evaluating the results of the developed system and system architectures methodologies will be explained in order to introduce concepts that will facilitate a better understanding of this project.

2.1 Building Representation

One of the important aspects in this thesis and in the development of an evacuation system is the ability to import a building representation, with the necessary information to construct a computational representation that allows the developed system to discover the best path to exit the building. Although there are a few ways to represent indoor spaces and their features, most approaches are graph-based, due to the facility of using well-known algorithms. A graph depiction typically has a set of nodes that define indoor spaces, such as rooms, corridors, staircases, elevators and so on, as well as a set of edges that represent the connection of areas that have a direct pathway between each other, thus creating a representation of a walkable network of indoor locations.

This section will describe and explain a few popular indoor mapping standards, exploring their main features to understand how they work and how they allow the representation of any area within the building's internal structure.

2.1.1 IMDF

The Indoor Mapping Data Format (IMDF) [2] is a specification developed by Apple Inc. and approved by the Open Geospatial Consortium (OGC) for representing indoor maps and related

information. Indoor Mapping Data Format (**IMDF**) conforms to the GeoJSON format [4], ensuring compatibility and transferability of the data. It is lightweight, mobile-friendly, and can be rendered on any device, operating system, or browser. It was designed with the main goal of providing a standardized format for indoor mapping data that can be used across different platforms and applications in large buildings such as airports, shopping, museums, and other large venues.

Some characteristics of **IMDF** are:

1. **GeoJSON File Format:** **IMDF** data is typically represented using the GeoJSON format, a well-known format for representing geographic data, providing a standardized way to represent indoor data in a JSON-based format.
2. **Structure:** Based on a hierarchical structure, where the top-level entity is the "IMDF Dataset", which represents a collection of indoor maps for a certain building. Within the dataset, it could have different layers of information, including floors, units, and features.
3. **Floors:** Multiple floors can be defined, and each floor represents a specific level within a building, each one represented by a unique identifier and containing information such as description, elevation, height, accessibility and connectivity. These last two attributes are important because contain level's accessibility features, such as whether it is wheelchair accessible, has ramps or elevators and how it is connected to other levels of the building, including information about stairs, elevators, escalators, or other vertical transportation methods.
4. **Units:** They represent physical spaces within a floor, such as a room, hallway, or other defined area. Each Unit can have attributes like name, category, and accessibility information and can be associated with other entities, such as features and openings.
5. **Features:** Represent doors, stairs, escalators, seating areas, and more objects or points of interest within a unit. Each feature has its own set of properties, such as its geometry, category, and accessibility information.
6. **Openings:** Openings in **IMDF** represent the connections or transitions between units, like doors, entrances, and stairs. Openings have properties that describe their geometry, type, and relationship to units.
7. **Geometry:** Support for various geometric shapes to represent the boundaries and positions of units, features, and openings, including polygons, polylines, and points.
8. **Metadata:** Possibility of including additional metadata for each entity, such as names, descriptions, phone numbers, URLs, and other relevant information.
9. **Relationships:** **IMDF** provides features to establish relationships between entities. For example, a feature can be associated with a specific unit, and an opening can connect two units. These relationships help define the connectivity and spatial relationships within the indoor space.

10. Extensions: **IMDF** supports the use of extensions to add custom attributes and properties to entities. This allows for the inclusion of domain-specific information that may not be covered by the core **IMDF** specification.

In summary, **IMDF** aims to provide an interoperable standard for indoor mapping data, enabling integration and exchange of indoor maps between different platforms and applications. In the last few years, it has gained significant adoption within the industry and is now supported by various mapping and navigation tools. Although Apple does not provide public API's for navigation with **IMDF**, there are third parties like MappedIn [5] and ArcGIS Indoors from Esri [9] who have developed paid software for navigation support. In Figure 2.1 we can see the development of a navigation network using ArcGIS Pro, a paid software that allows the extraction of the represented information from an **IMDF** document.

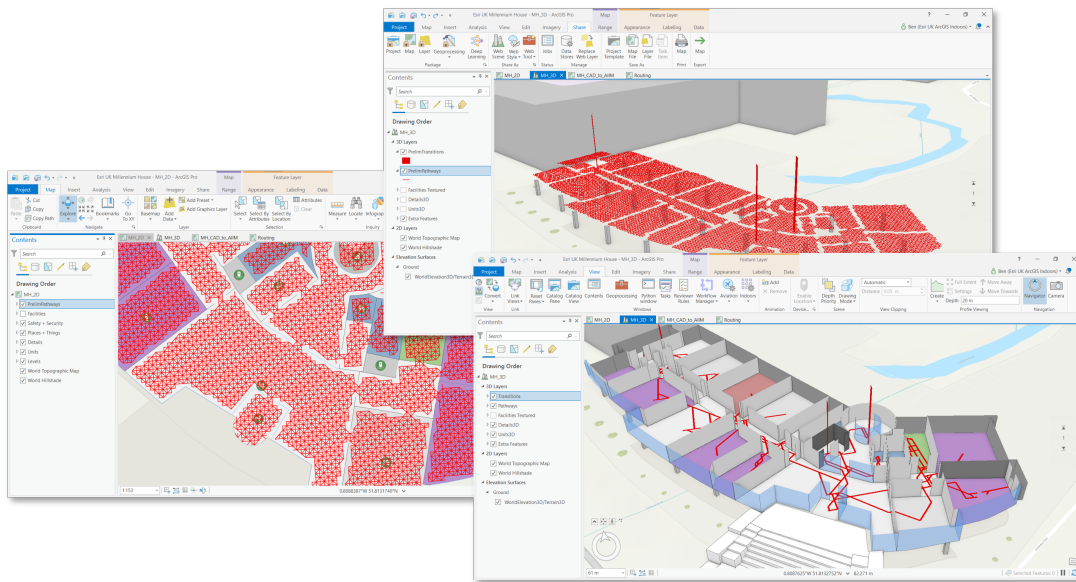


Figure 2.1: ArcGIS Indoor Routing, retrieved from [10]

2.1.2 IndoorGML

Indoor Geography Markup Language (IndoorGML) [19] is an Open Geospatial Consortium (**OGC**) standard, published in 2016, defined as an application schema of **OGC** Geographic Markup Language (**GML**) for a data model and XML-based format, with the goal of representing indoor spatial information, tailored to address the complexities and intricacies of indoor spaces.

IndoorGML aims to provide a comprehensive framework that can represent different elements of indoor spaces, including buildings, floors, walls, doors, rooms, corridors, stairs, and more, allowing for the modeling of both geometric and semantic information related to indoor features. In [6] the authors make a significant contribution to the understanding of IndoorGML, exploring the vital topic of the subdivision of indoor spaces, and providing valuable insights into the methods and techniques employed to represent complex layouts and structures in indoor environments.

They discuss the nuances of the IndoorGML data model, emphasizing its ability to capture and define the intricacies of indoor spaces, which has served as a valuable resource, providing essential knowledge to better understand how IndoorGML works.

Some features of IndoorGML are:

1. **Cell Geometry:** Indoor spaces are represented as a set of cells, each one having its unique Identifier (ID) and information that describes them such as name or tags. The two most important topology aspects are adjacency and connectivity between cells. Cells can be adjacent to each other, meaning they can share boundaries, but cannot overlap them and connectivity describes how cells are connected to each other, including information about how users can move from one cell to another, using doors, corridors, stairs, or other transitions.
2. **Semantics and Metadata:** IndoorGML contains semantic modeling capabilities, allowing for the attachment of metadata and descriptive information to indoor features, such as room types, functions, and other relevant information or attributes.
3. **Integration with Other Standards:** Possibility of integration with other geospatial standards, such as CityGML and SensorML, enabling indoor spatial data to be part of broader geospatial models and sensor networks.
4. **Multi-Layer Space Model:** An indoor space may be interpreted in several ways depending on viewpoints. For example, a corridor can be a cell for walking pedestrians but a cell in this corridor can also be seen as an area of actuation of a sensor. The same physical area can have different viewpoints, leading to two different configurations of an indoor layout, where each configuration is called a space layer. With the multi-layer space model of IndoorGML, we can integrate multiple space layers via inter-layer connections, which is a very powerful feature of IndoorGML.
5. **File Validation:** Since IndoorGML is based on XML, tools like `val3dity` [14] allow validating the integrity of files against the defined XML schema provided by the creators of IndoorGML [19].

The developers responsible for maintaining this standard, STEMLab from Pusan National University, provide some open-source tools to support the creation, visualization and validation of IndoorGML data. These tools are designed to facilitate the process of representing the indoor characteristics of buildings in IndoorGML, by providing user-friendly interfaces and functionalities tailored for working with IndoorGML.

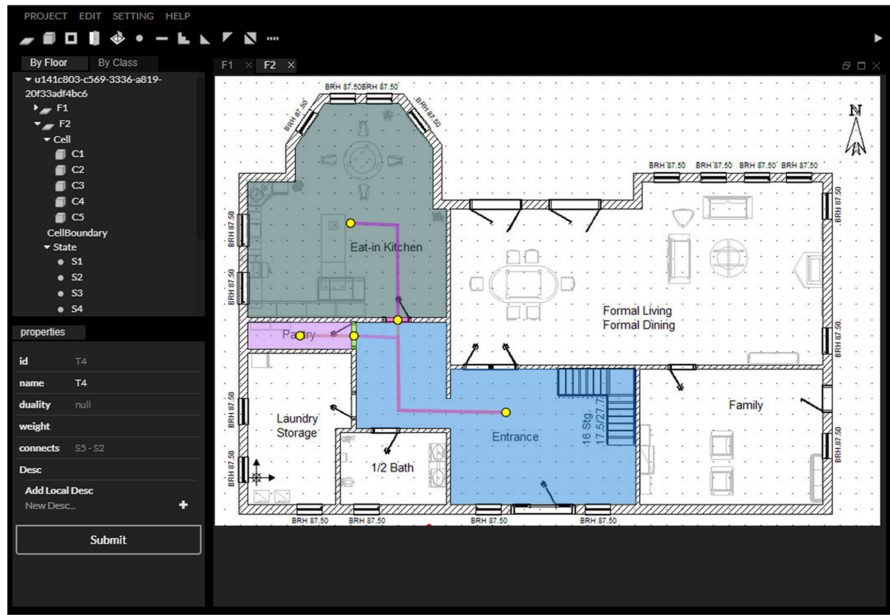


Figure 2.2: InEditor Software

Image source: <http://stem.pusan.ac.kr/web/project/InEditor/>

One of the most used tools is InEditor [22], a web-based editor for drawing and creating OGC IndoorGML data. This software provides a graphical user interface Graphical User Interface (GUI) where users can interactively define and modify indoor spaces and their attributes. InEditor includes features such as drawing tools for creating floor plans, tools for adding and editing indoor objects, and support for defining semantic information.

2.2 System Architecture

This section presents and explains the different types of system architectures, exploring their structural differences and respective advantages and disadvantages, with the intent of helping to choose the best architecture for the system to be developed.

The choice of system architecture is one of the most important steps in the development of this work. There are three main types of architecture designs: centralized, decentralized, and distributed. They all have advantages and disadvantages for different purposes. Fault tolerance, maintenance costs, scalability and implementation effort are some characteristics that differentiate them and that we must analyze.

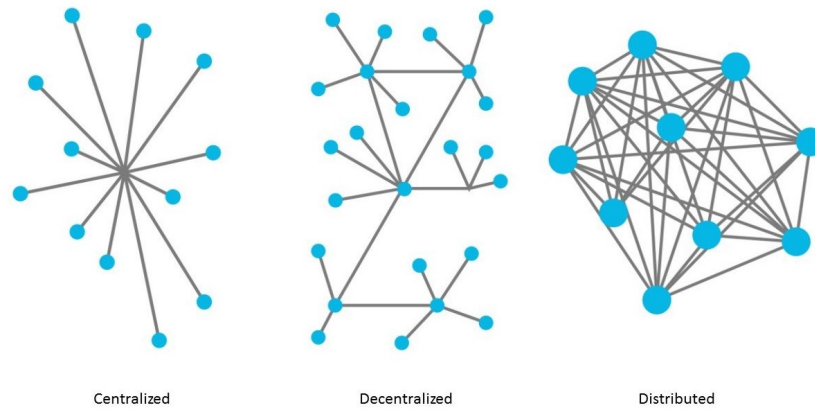


Figure 2.3: System Architectures Overview

2.2.1 Centralized Systems

Centralized systems are probably the most intuitive and easy to understand type of architecture, using a client/server architecture where client nodes connect directly to a central server. This is the most commonly used type of system in many situations where a client sends a request to a company server and receives the response. These systems have important limitations. If the central node fails, the entire system fails, because the client nodes are not able to obtain data, and this type of approach has difficult server maintenance, because there is only one server and it must not be down for maintenance, although a solution with multiple replicated servers can solve this problem. Relating to the proposed work, a central server, that receives the environment values read by the sensors, would be developed. This server would play a central role, in knowing the building plant and calculating, in case of an emergency, the safest path to evacuate the building from different locations, communicating the evacuation path information to devices that can transmit this information to the users.

2.2.2 Decentralized Systems

In decentralized systems, there is no central node. Instead, there are multiple authoritative nodes, each of which has a copy of the data and serves a subset of the end users. Even though it still is vulnerable to server crashes, a decentralized system tends to be more tolerant to faults because if one authoritative node fails, there are still other nodes that can handle user requests. Relating to the proposed work, it would not be much different from the Centralized System approach. The main difference would be the existence of various nodes that can compute the safest path to evacuate the building and transmit this information to a certain number of devices that will transmit this information to the users.

2.2.3 Distributed Systems

In a distributed system, there is a network of synchronized nodes that communicate between them, and every node makes its own decisions. There is not a central unit that commands the system, and the final output of the system is an aggregation of decisions individually taken by different nodes. Relating to the proposed work, all deployed nodes communicate between them and have computing power, meaning that they can calculate the safest path to evacuate the building. Basically, several nodes are always receiving environment values read by sensors, and these nodes communicate with each other to exchange information and be able to have a global view of everything that is happening in the building, allowing them to calculate the safest path when an emergency occurs, transmitting this information to devices that indicate to the users the right path to follow. With this type of approach, we need to guarantee that every node has the same information to ensure consistent decisions, which brings us to the problem of reliable data transmission between nodes.

2.3 Search Algorithms

One of the most important components, with strong implications for the system's performance, is the search algorithm used to calculate the appropriate evacuation paths. In normal conditions, there is no emergency occurring, and the safest route to the exit the building is usually the shortest route. However, when an emergency occurs this may not be the ideal path, because, for example, the shortest path may be affected by the hazard, for example, a fire source, and the user will not be able to use this path safely. Thus, the evacuation algorithm, through sensor monitoring, must take into account environmental variables and guide the users through a safer route to the exit. Usually, the building navigation path representation is based on a graph, where edges between nodes represent direct connections from one location to another. Graphs can be directed or undirected, and unweighted or weighted (if each transition can have an associated cost). In an undirected graph, edges are bidirectional, whereas in a directed graph there may be a direct connection between node A to node B but not from B to A. In emergency systems, the value associated with a connection from A to B is usually related to the physical distance and how long it takes to travel from A to B, danger level, connection congestion or even a more complex formula that combines some of these metrics to define a safety value of an edge. In this section, some of the most common algorithms that are able to solve the shortest path problem will be explored and described, in order to understand which one better fits the proposed system.

2.3.1 Dijkstra's algorithm

Dijkstra's algorithm is a well-known algorithm for finding the shortest path tree from a root node, in a weighted graph, developed by Dutch computer scientist Edsger W. Dijkstra in 1956. This algorithm maintains a set of "visited" nodes and a set of "unvisited" nodes, starting at the

source node, which is assigned a distance of zero and a distance of infinity to all other nodes. The algorithm then repeatedly selects the unvisited node with the smallest distance and "visits" it, adding this node to the set of visited nodes. For each visited node, the algorithm updates the distances of its neighboring nodes that are still unvisited. Specifically, it computes the distance to each neighboring node as the sum of the distance to the current node and the weight of the edge connecting the current node to the neighboring node. If this computed distance is less than the current distance assigned to the neighboring node, then the algorithm updates the distance to the computed value, terminating when the destination node is added to the set of visited nodes or when there are no more unvisited nodes. Dijkstra's algorithm is a greedy algorithm, because at each step it chooses the node with the smallest distance, which ensures that the algorithm always selects the optimal path to a node. The algorithm works correctly for graphs with non-negative edge weights, and should be used when it's necessary to find the shortest paths from one source vertex to all other vertices in a graph, otherwise the use of the Bellman-Ford algorithm is recommended.

The time complexity of Dijkstra's algorithm is $O(|E| + |N| * \log(|N|))$, where $|N|$ is the number of nodes in the graph and $|E|$ is the number of edges.

2.3.2 Floyd-Warshall algorithm

The Floyd-Warshall algorithm, developed by Robert Floyd and Stephen Warshall in 1962, is a dynamic programming algorithm for finding the shortest paths between all pairs of nodes in a weighted graph, based on a matrix of distances between all pairs of nodes that is initially set to the weight of the edges in the graph, or infinity if there is no edge between two nodes. The algorithm then proceeds to iteratively update the matrix, considering, at each iteration, all pairs of nodes (i, j) and a third node k . It then checks whether the path from i to j through k is shorter than the current shortest path between i and j . If the path through k is shorter, the algorithm updates the distance matrix accordingly.

The basic idea behind the algorithm is that if there is a shorter path between two nodes that goes through a third node, then the shortest path between the original two nodes should also go through that third node. By iteratively considering all possible intermediate nodes, the algorithm can find the shortest paths between all pairs of nodes in the graph.

The time complexity of the Floyd-Warshall algorithm is $O(N^3)$, where N is the number of nodes in the graph, and this algorithm should be used when it's necessary to find the shortest paths between all pairs of vertices in a graph and when negative edge weights are present (but there are no negative cycles).

2.3.3 Bellman-Ford algorithm

The Bellman-Ford algorithm, presented by Richard Bellman and Lester Ford in the 1950s, is an algorithm for finding the shortest paths between a single source node and all other nodes in a weighted graph, even if it contains edges with negative weights, whereas Dijkstra's algorithm can only be applied in positive weight values [23].

Initially, the distance from the source to all other nodes is set to infinity and then the algorithm iteratively "relaxes" those estimates by finding alternative paths that are shorter than the previously overestimated routes, repeating this process for a total of $n-1$ iterations, where n is the number of nodes in the graph. After $n-1$ iterations, the algorithm has guaranteed to find the shortest path to all nodes that can be reached from the source node, as long as the graph does not contain a negative-weight cycle.

A negative-weight cycle is a cycle in the graph that has a negative total weight, making the shortest path problem ill-defined, as it is always possible to find the shortest path by repeatedly traversing the cycle.

The time complexity of the Bellman-Ford algorithm is $O(nm)$, where n is the number of nodes and m is the number of edges in the graph. In the worst case, the algorithm may have to perform $n - 1$ iterations, each of which involves checking all m edges.

2.4 Databases

Frequently, evacuation systems need to save information related to building characteristics, configuration values, sensor location and values, as well as data relating to navigation in the building.

A Database (DB) is an organized collection of structured information, designed to efficiently store, retrieve, and manipulate large amounts of information, usually stored in a computer system and generally controlled by a Database Management System (DBMS).

There are two main types of databases, relational and non-relational, described below, but they share the core purpose of providing a systematic way of storing and organizing data, ensuring data integrity, and enabling efficient retrieval and manipulation of information, commonly organized into tables with rows and columns. They can store a wide range of data types such as text, numbers, dates, images, and more.

2.4.1 Relational Databases

Relational databases are the most common type of database, because this type of database organizes and stores data in a structured manner using tables, relationships, and a set of

predefined rules.

A table is the fundamental unit of organization and consists of a set of rows (records) and columns (fields/attributes). Each table represents a specific entity or concept in the database schema. For example, a database from a university, might have tables for students and courses, with the first one storing information from students such as ID, address or age and the second storing information about each course, like the number of students, course name and other important information.

Another fundamental aspect is the Keys, used to establish relationships between tables. The primary key is a unique identifier for each row in a table, ensuring that each record can be uniquely identified. Foreign keys are used to create relationships between tables by referencing columns of other tables. This mechanism helps maintain data integrity and enables efficient data retrieval through these relationships.

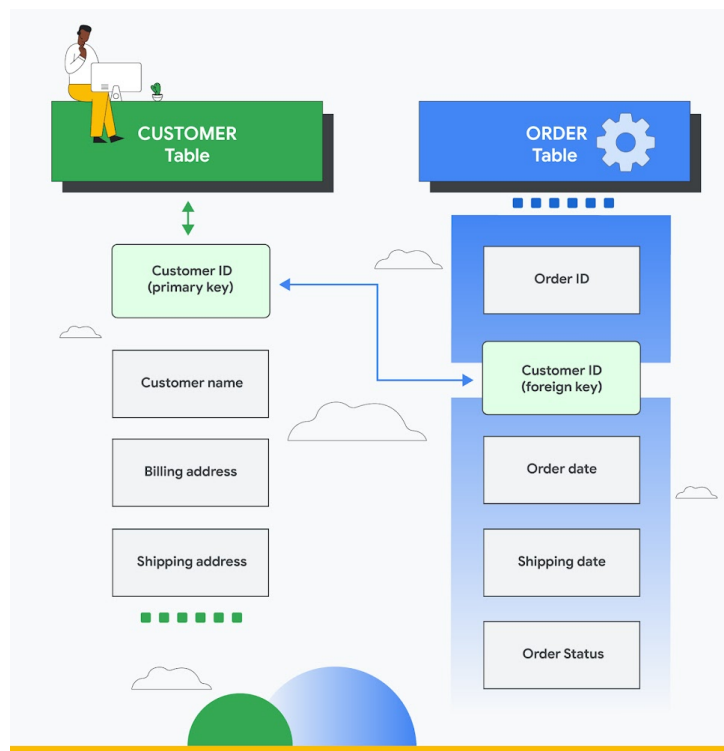


Figure 2.4: Example relational database structure, retrieved from [13]

With relational databases, Structured Query Language (SQL) queries are commonly used in order to get, insert, modify, delete or link information in tables, with the major advantages of being simple and intuitive.

There are a large number of relational databases like MySQL, an open-source management system that maintains high availability, scalability and performance; PostgreSQL, also an open-source database manager that retains a high level of performance and flexibility, which is useful in large read/write speeds, especially through big amounts of data; And finally MariaDB, an

open-source community-developed relational database that is a fork of the MySQL database and offers compatibility with MySQL, making it a popular alternative. MariaDB provides features such as high performance, scalability, and robustness, along with additional enhancements and optimizations. Relational databases are preferred when data integrity, complex queries, and transactional consistency are necessary.

2.4.2 Non-Relational Database

Non-relational databases, also known as Not Only SQL (**NoSQL**) databases, are a type of database management system that does not rely on the traditional tabular, relational model of data organization used in relational databases. Instead, they provide a flexible and scalable approach to handling vast amounts of unstructured or semi-structured data.

There are four main types of NoSQL Databases, as shown in 2.5:

1. Document-based databases: Store data in JavaScript Object Notation (**JSON**), Binary JSON (**BSON**), or Extensible Markup Language (**XML**) documents.
2. Key-value stores: Each element in the database is stored in key-value pairs and the data can be retrieved by using a unique key allotted to each element.
3. Column-oriented databases: Store the data in columns instead of rows, meaning that when we want to fetch information on a small number of columns, those columns can be read directly without consuming memory with the unwanted data.
4. Graph-based databases: Store the data in the form of nodes in the database, connecting these nodes through links/relationships.

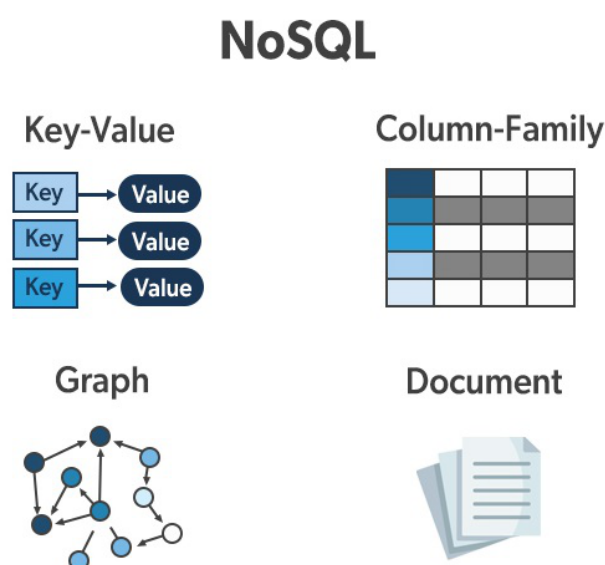


Figure 2.5: Types of NoSQL Databases, retrieved from [12]

In summary, Non-relational databases offer flexibility, scalability, and high performance, making them suitable for scenarios where flexibility and horizontal scaling are essential and where traditional relational databases may not be the most efficient solution. This means that non-relational databases are thus ideal for systems that need to handle a vast amount of distinct types of data or systems that need to save information that may be modified constantly such as big data, real-time applications, and distributed computing.

Two of the most popular NoSQL databases are MongoDB and Redis. MongoDB is a cross-platform open-source document-oriented database that stores data in JSON-like documents in dynamic schemas. Redis is an in-memory, key-value store that supports various data structures, including strings, lists, sets, and sorted sets, gaining popularity for its exceptional performance and is often used for caching, session management and messaging systems.

2.5 Network Simulation and Emulation Tools

To facilitate the process of testing the developed system and due to the cost of deploying a complete system in a real building during an initial development phase, simulation and emulation tools are usually used to understand whether the system is ready to be tested in real buildings. These tools are commonly programs that allow the user to build a virtual network with common components such as computational units, routers, switches and establish communication between them. It is important that this type of software provides a way to simulate problems with latency, congestion or lack of connection in the network, to test the system when facing these common problems that occur in emergency scenarios.

In this section, we will find two types of programs, emulators and simulators. A network emulator is a specialized tool or software application used to replicate and simulate the behavior of real-world networks in a controlled environment. Its primary purpose is to mimic the characteristics, conditions, and challenges that networks may encounter during operation. A network simulator, on the other hand, is a software tool or application that models the behavior of a network by using mathematical and logical models. It allows users to design, simulate, and analyze the operation of networks without the need for physical hardware.

These tools will be useful in this work because will allow us to test the developed system in different conditions quickly, without the need to install the entire system in a building, which would bring higher costs and time requirements.

2.5.1 GNS3

GNS3 (Graphical Network Simulator-3) is an open-source network emulator and virtualization software that allows the simulation and testing of complex network topologies. It is provided a **GUI** that enables users to design, configure, and interact with virtual networks using a wide range of technologies and networking devices such as routers, switches, and firewalls, using real

operating systems (OS) images, including popular vendors such as Cisco, Juniper or MikroTik.

GNS3 leverages virtualization technologies, such as QEMU (Quick Emulator) and VirtualBox, to run the emulated network devices and the GNS3 GUI allows to visually design network topologies by dragging and dropping network devices onto a canvas and configure the devices using the respective OS images and Command-Line Interface (CLI).

GNS3 allows the creation and testing of various network scenarios, such as routing protocols, network security configurations, and network performance testing and can integrate with network monitoring and management tools, such as Wireshark and network analyzers, to capture and analyze network traffic within the virtualized environment, allowing for deeper network troubleshooting and analysis makes GNS3 a powerful platform for network simulation and testing.

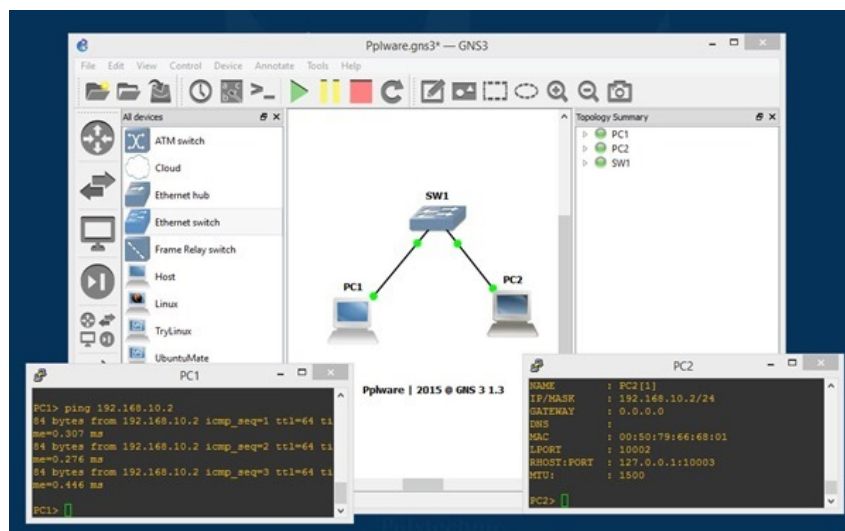


Figure 2.6: GNS3 Graphical User Interface, retrieved from [18]

2.5.2 Mininet

Mininet is a network emulator that allows the creation of virtual networks that behave like real networks. One of the key features of Mininet is its scalability, since it is built on top of Linux virtualization technologies, such as network namespaces and Linux Containers (LXC). Mininet provides lightweight and isolated network environments on a single physical machine, allowing users to test and develop large-scale network applications and protocols with hundreds or even thousands of nodes.

This software allows users to create network topologies by defining switches, hosts, and links between them. The topology can be created using a simple Python-based API, which allows users to programmatically create and manage the network. It also provides a simple graphical user interface, MiniEdit, that allows the creating of network topologies by dragging and dropping switches and hosts onto a canvas and connecting them using links.

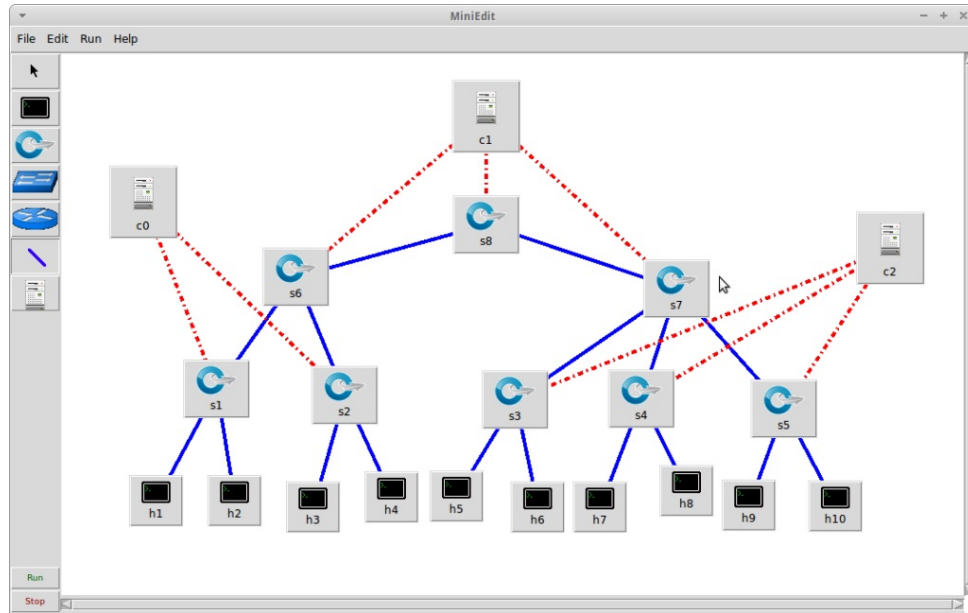


Figure 2.7: MiniEdit GUI

Once the network topology is created, Mininet can emulate traffic in the network using various traffic generation tools, such as ping, iperf, and hping. This allows users to test and develop network applications and protocols in a controlled environment.

Mininet is commonly used for Software Defined Networks development because it supports the OpenFlow protocol, which allows users to program the behavior of network switches. This enables users to test and develop OpenFlow applications and controllers. For example, a user can program a switch to forward traffic to a specific port based on the destination MAC address or IP address.

In summary, Mininet is a powerful network emulator that allows users to create virtual networks and test and develop network applications and protocols in a controlled environment. Its features include network topology creation, traffic generation, OpenFlow support, scalability, and integration with SDN platforms.

2.5.3 EVE-NG

Emulated Virtual Environment-Next Generation (EVE-NG) is a network emulation platform that allows designing and testing network topologies and configurations, providing to users the possibility of emulating different network devices, such as routers, switches, firewalls, and servers to create complex virtual network environments. EVE-NG offers 2 versions, a community edition and a professional edition with more features and professional support.

Some key features and functionalities of EVE-NG include:

1. Network Devices: Emulation of network devices from well-known vendors, including Cisco,

Juniper, Fortinet, and more. Allowing users to simulate real-world network scenarios by creating virtual instances of these devices, configuring them and mixing and matching devices from different vendors on the same network topology to test interoperability between different network technologies.

2. **Network Topology Design:** Users can design complex network topologies by connecting virtual network devices and creating custom network configurations to test different network designs before deploying them in the real world.
3. **Virtual Labs and Experiments:** EVE-NG provides a virtual lab environment where users can perform experiments and test different network configurations safely, because the software offers features like snapshots and restore points, allowing the user to save and revert to specific states, which makes it easier to experiment new setups.
4. **Integration with External Tools:** Integration with external tools and applications, such as Wireshark and GNS3, to enhance network analysis and monitoring capabilities. It also enables users to manage and configure emulated devices remotely due to the compatibility with remote access protocols like SSH and Telnet.
5. **Active Community:** Support of a community that shares network topologies, configurations, and labs, allowing users to share knowledge and collaborate with others and keep improving the product.

In summary, EVE-NG is one of the most powerful emulation tools for networks, providing several advanced tools that require deep knowledge and a slow learning curve to be able to take advantage of all its capabilities.

2.5.4 NS-3

NS-3 is an open-source discrete-event network simulator [17] which is widely used for research and development in the field of networking, representing a platform for simulating and studying the behavior of complex network systems, protocols, and applications.

Some of the key characteristics of the NS-3 are:

1. **Modular and Extensible:** Designed with a modular architecture, which allows users to easily extend and customize the simulator with the help of a wide range of models and modules for different network components, protocols or applications.
2. **Support for various network types and technologies:** Provides a wide range of network types, including wired, wireless, and ad hoc networks, making available models for the most used network technologies, such as Ethernet, Wi-Fi and LTE.
3. **Network Protocols:** Ns-3 includes implementations of network protocols, such as TCP/IP, OSPF or BGP routing protocols, transport protocols like TCP and UDP, and application protocols (e.g., HTTP, FTP).

4. Visualization and Analysis Tools: It is available tools, such as graphs, plots, and statistics metrics for visualizing simulation results and analyzing network performance.
5. Large Community: NS-3 is an open-source software with an active community of users that keep improving the software and provide support. The project also provides documentation, tutorials and examples to help users get started with ns-3.

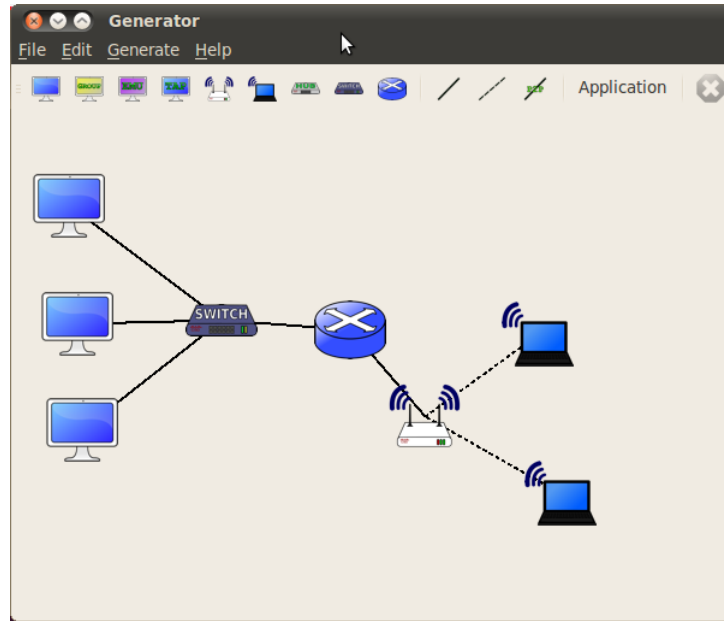


Figure 2.8: NS-3 Network Topology Diagram Tool, retrieved from [18]

2.5.5 Netem

Netem is a network emulation tool, part of the Linux kernel's Traffic Control (tc) subsystem, that allows users to simulate various network conditions and behaviors, commonly used to assess the performance, reliability, and robustness of networked applications, enabling users to introduce artificial delays, packet loss, reordering, and other network impairments to emulate real-world network conditions.

Some of the key features and capabilities of Netem include:

1. Delay: Introduce delays in network traffic, simulating latency in the network, useful for evaluating applications, such as real-time communication systems, that are sensitive to network delays.
2. Packet Loss: Emulate packet loss by dropping a certain percentage of packets or following specific loss patterns.
3. Reordering: Simulate packet reordering, where packets arrive at the destination out of order, useful for testing streaming apps, where packet ordering is an important task.

4. **Corrupt and Duplicate Packets:** Introduction of corrupted and duplicated packets to test applications' error-handling and recovery capabilities.
5. **Bandwidth Limitation:** Limit the bandwidth available to a network interface, allowing to test the performance of applications under constrained network resources.

In conclusion, Netem is commonly used to simulate different network scenarios and evaluate the behavior of networked applications, helping in identifying potential issues, optimizing network performance, and testing the reliability of systems under adverse network conditions, like in emergency scenarios. Netem represents a powerful tool when combined, for example, with Mininet, allowing the construction of an entire network and testing this network under the described problems.

Chapter 3

State of the Art

The preceding chapter discussed critical aspects related to evacuation systems, including various architectural aspects, indoor mapping standards and search algorithms but also softwares that allows for testing these systems in the development phase. This analysis provided us with a comprehensive understanding of the fundamental factors that make up an indoor evacuation system. This chapter will explore the state of the art in this field, discussing some of the more relevant previous research studies and analyzing the different metrics and methodologies between these works and the divergences between each other.

3.1 A Distributed Decision Support System

One of the most important works that inspired this thesis was proposed by Avgoustinos Filippoupolitis and Erol Gelenbe in June 2009 [11]. The authors proposed a decision support system based on a distributed architecture to calculate evacuation routes in real-time, using distance and intensity of the hazard as metrics to compute all available evacuation paths.

3.1.1 Assumptions

There are many assumptions that we need to understand before exploring the system:

1. The building layout is already known and the decision support network is already pre-installed and deployed in the building's internal structure.
2. The existence of several Decision Nodes (DN): these are nodes, installed in specific locations inside the building, that do not need to have much processing power or storage, since their task is only to compute the direction that should be followed by individuals, during an emergency, towards the best available exit.
3. The direction to follow over the building to reach an exit is obtained by smart panels that are deployed in the building or by a wireless device that each evacuee has.

4. The existence of a network of sensor nodes capable of providing real-time information about the conditions inside the building to the Decision Nodes, such as the temperature of a space or the presence of smoke.

3.1.2 Graph Representation of the Building

In the solution proposed, an example building layout is used to create a graph to representation of it, as shown in 3.1. The vertices of the graph represent possible users locations along the building and the links represent physical paths between these vertices that users can use to move around the building and find an exit.

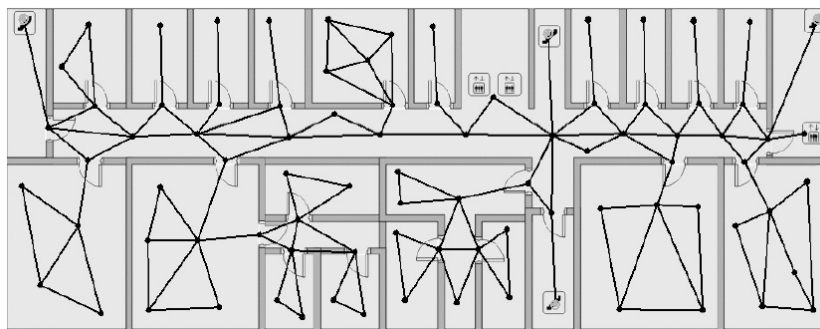


Figure 3.1: Graph of building representation

To construct this navigable graph, two values are used for calculating the weight of each link, the navigable distance between a pair of nodes, represented by $l(i, j)$, and the hazard intensity, represented by $H(i, j)$, obtained by the sensor that controls the transition between nodes i and j and that is initialized in 1, increasing with the intensity of the observed hazard. Using these two values, the formula proposed by [11] to calculate the actual weight of each link between two decision nodes is represented by:

$$L(i, j) = l(i, j) * H(i, j)$$

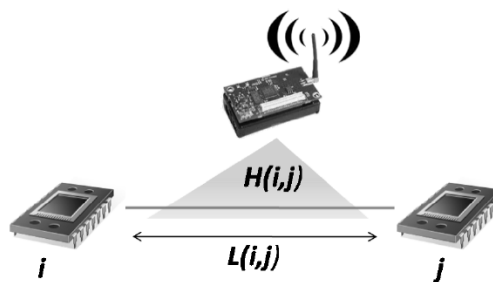


Figure 3.2: Example of sensor and link between two decision nodes

3.1.3 System Architecture

This work uses a distributed architecture, where each node stores enough information to be able to calculate the safest path to exit the building and to communicate to the user the next decision node to follow, which means that there is no central unit that processes all the information.

As we can see in figure 3.3, several Decision Nodes are installed in the building, and collect information from Sensor Nodes that are installed in the links that connect Decision Nodes.

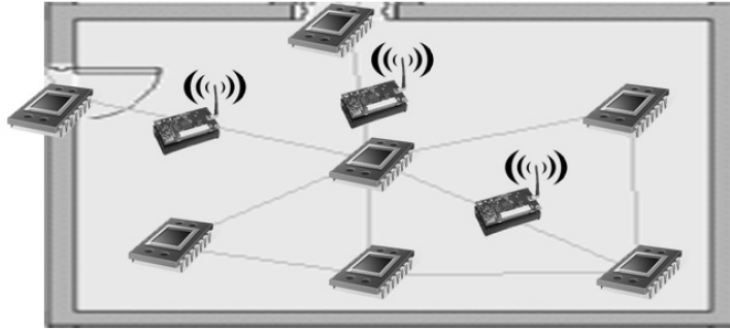


Figure 3.3: Sensors and Decision Nodes deployed in the building

A decision node x stores the following information:

1. The actual weight, obtained by ‘effective length’ L , of each transition of x .
2. The effective length of all neighboring transitions of x to an exit e .
3. The effective length of the shortest path from x to an exit e .
4. The recommendation for the next best decision node.

3.1.4 Safest Path Result

Since the decision node network is already deployed before any hazard event happens, and, in normal conditions, the hazard intensity between nodes is equal to 1, meaning that the effective length of a transition between two nodes is equal to the physical distance between those nodes, each decision node will already have stored the information of the shortest path for an available exit. When an emergency occurs, the algorithm proposed by [11] will be periodically executed in each decision node, calculating the effective length of the path from its location to the exit and sending this information to each of its neighbors. It will update the effective length of the shortest path to the exit and choose the next decision node to follow that minimizes that effective length, finding in this way the route with the total minimum sum of all its link’s weight (calculated with the formula shown above).

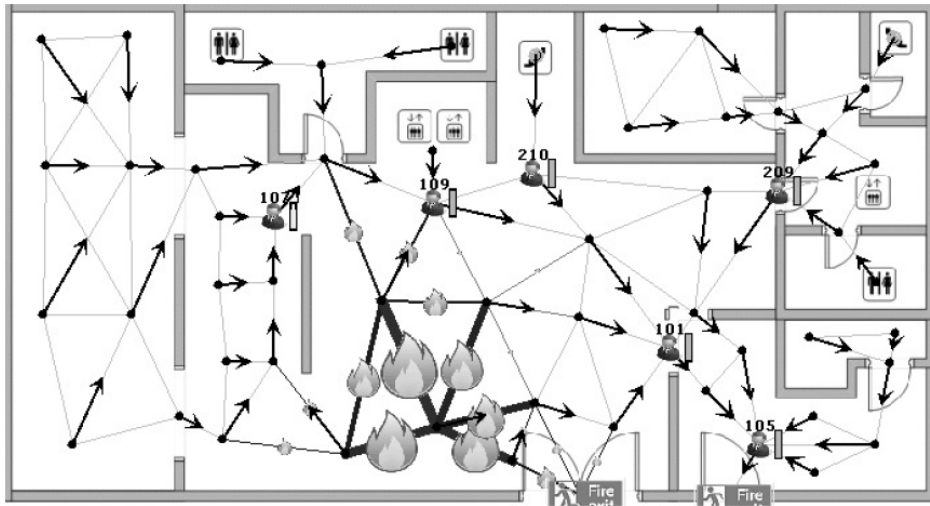


Figure 3.4: Example safest path calculation

3.1.5 Conclusion

This work presented a decision support system based on a distributed architecture, with the help of decision nodes and sensors deployed along the building, which avoids a singular point of failure when compared to other approaches, since a single point of data collection, processing and re-transmission, in the case of a failure, causes the whole system to fail, which is unacceptable in emergency scenarios. Each decision node computes the best direction towards the exit using only local information and communicates the directions of the next Decision Node to follow to the evacuees via Smart Panel Indicators or Wireless Devices that they carry. This type of architecture propose allows expansion of the system when necessary, by simply increasing the number of decision nodes and Smart Panels or individual devices, according to the size of the building without being limited by the computational power of a single machine.

In this work is not mentioned what happens when a Decision Node fails and how this affects the system and consequently the the effectiveness of the evacuation process.

3.2 A Smart Evacuation Guidance System for Large Buildings

One of the main reference and recent research studies was proposed by Van-Quyet Nguyen, Huy-The Vu, Van-Hau Nguyen and Kyungbaek Kim in 2022 [16]. Some of the authors had previously proposed a first approach to this theme in 2019 [15], implementing the system three years later. They propose a flexible and scalable evacuation system for large smart buildings with multiple levels of computational support, using a distributed algorithm for finding effective evacuation routes with an LCDT (Length-Capacity-Density-Trustiness) weighted-graph model, which is built upon the disaster information and building conditions. They also proposed a

caching strategy that leverages the efficient route part(s) on the LCDT weighted graph in order to improve the performance of dynamic routing in large buildings.

3.2.1 System Architecture

The proposed system architecture is shown in figure 3.5 and operates based on the IoT data obtained from sensors and cameras. A network of smart indicators on each floor is presented as a sub-graph, where a node represents a smart indicator, and an edge represents a route segment between two indicators that can be used by evacuees to exit the building. In this solution there are three main modules:

1. **Smart Indicators:** This component is responsible for calculating the weight of a segment, that represents the disaster condition on the corridor where it is located, and showing the direction to guide evacuees to a safe exit. The Smart Indicator is able to calculate the weight of his segment with the help of temperature and smoke sensors, and with a camera collecting images. These images are processed by a Density Calculator module, allowing us to understand the density of people in the corridor, using a pre-trained Convolutional Neural Network, [24]. Another important role of this component is to show the direction that users must follow, calculated by the Smart Guidance Agent. To obtain this information, there is a Guidance Director module that every 5 seconds gets information from an Application Programming Interface (API), checking for new information sent by the SGA.
2. **Smart Guidance Agents (SGAs):** They are responsible for controlling the Smart Indicators of a specific region, e.g., a floor, constructing a weighted sub-graph based on the structure of a network of smart indicators in the given floor and the weights of segments sent by smart indicators. It also contains a Local Evaluator module that is responsible for finding, using the Dijkstra algorithm, efficient routes from every smart indicator to stairs, sending this information to the Smart Coordinator, which has information from all areas and can calculate the best path to exit the building. To discover the safest evacuation path for each smart indicator, the SGAs use the Evacuation Routes Selector module, which has information received from the Smart Coordinator, by combining the weight from smart indicators located at stairs and the weight among stair nodes of several floors sent by Partial Evaluator module.
3. **Smart Coordinator (GC):** The main objective of this component is to provide a global view information in order to help the SGAs find the best route that avoids hazardous and high-congestion corridors. It first builds a weighted cross-graph based on the structure of network stair-nodes of the several areas, with the help of the weights among stair-nodes sent by the Local Evaluators of each area. Then it finds the shortest path from a stair node to other stair nodes in the lower floors for every weighted cross-graph and transmits this information to the Evacuation Routes Selector from each SGA, to be obtained by the smart indicators via the available API.

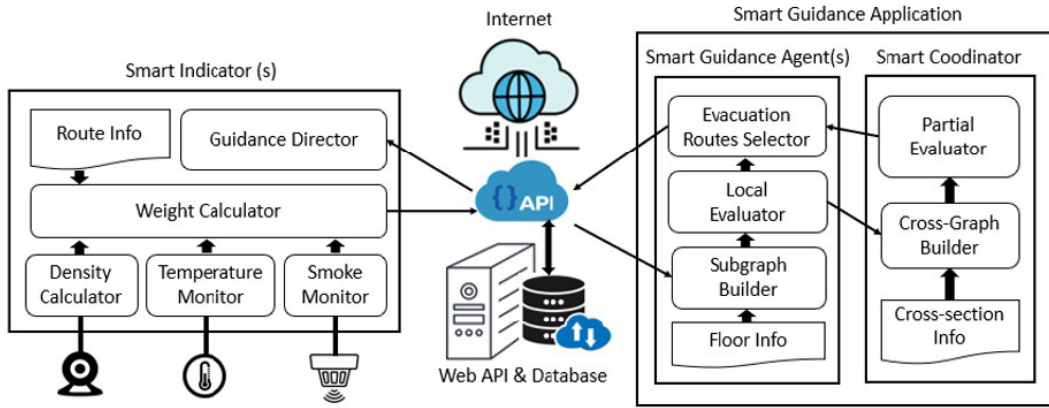


Figure 3.5: Overview of System Architecture

3.2.2 Smart Indicators Design

As shown in Figure 3.6, the authors propose the implementation of the Smart Indicators using a Raspberry Pi with internet connection via Wi-Fi, an Universal Serial Bus (USB) camera for collecting images, used by the Convolutional Neural Network (CNN) to measure the people density in the corridor, and sensors like DHT11 for Temperature and MQ135 as gas sensor for detecting the presence of smoke.

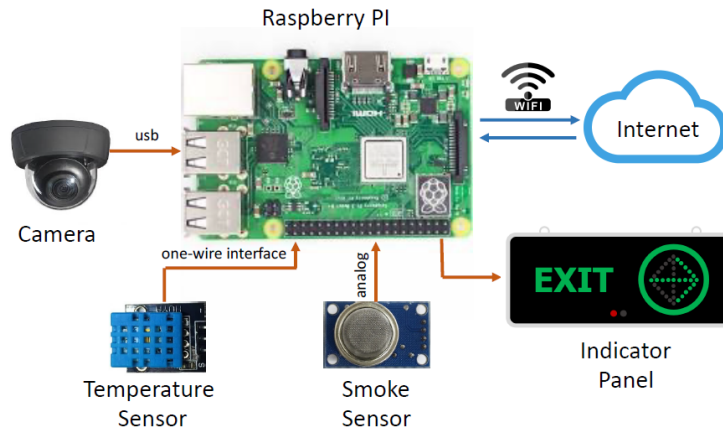


Figure 3.6: Smart Indicator Components

3.2.3 LCDT-Based Weighted Graph for Finding Evacuation Routes

This paper considers a network of smart indicators in the building as an undirected graph $G = (V, E)$, where V is a finite set of nodes, and E is a finite set of edges. A node in graph G is represented physically by a smart indicator(Indicator Node) and each edge in the graph is a path segment in the building between two adjacent smart indicators. As described in the previous

section, SGAs will calculate the weight of each edge and this value indicates how much the disaster affected each route segment. The higher the value of the weight, the longer evacuation time is necessary. This value is calculated by SGAs taking into account four factors: segment physical length (fixed value) and width, hazard intensity and people density. Assuming that an evacuee needs a minimum lane space $X * Y$ (e.g., 1 square meter) for moving, the capacity of a segment between two adjacent smart indicators v_i and v_j at a physical length $L_{(i,j)}$ and width $H_{(i,j)}$ is obtained using the following formula:

$$C_{(i,j)} = (L_{(i,j)}/X) * (H_{(i,j)}/Y)$$

The final equation to calculate the weight of each edge is given by the formula presented below, where $T_{(i,j)}$ is a value between 0 and 1 representing the trustiness of the segment: 0 means that the segment was seriously affected and 1 means that the segment is in normal conditions. The value of $T_{(i,j)}$ is defined with the help of the information provided by smoke and temperature sensors. The density of people in the segment, $D_{(i,j)}$, is obtained by the CNN based approach of people counting. The final formula for the weight is:

$$W_{(i,j)} = \frac{(L_{(i,j)})}{T_{(i,j)} * (C_{(i,j)} - D_{(i,j)} + 1)}$$

3.2.4 Conclusion

This approach presented a design and implementation of an emergency evacuation system that uses a dynamic evacuation routing approach and smart indicators, based on a distributed architecture with multiple layers of computing that provide a routing solution with partial information, from several regions of the building, which represents the hazard intensity and the crowd congestion in each region. Although the computational decisions are distributed by different components, it still has a singular point of failure, the Smart Coordinator, which is responsible for having a global view of the building and joining information obtained by each Smart Indicator to find possible routes to exit the building, meaning that if the Smart Coordinator fails, the system will fail. The LCDT-Based Weighted Graph was also an improvement over previous works because it can handle the congestion of corridors, preventing all evacuees from using the same corridor and not being able to be evacuated in useful time, with the help of real-time images, captured by a camera and processed by a CNN model.

3.3 Emergency Evacuation using Wireless Sensor Networks

Another research study explored for this thesis, proposed by Matthew Barnes, Hugh Leather and D. K. Arvind from School of Informatics, University of Edinburgh in 2007 [3], consists of a proposal for a distributed evacuation system using Wireless Sensor Networks. This approach

uses evacuee movement times and hazard propagation times as metrics for calculating evacuation paths for an emergency exit.

3.3.1 Building Representation

In this work, the building's environment is represented by two weighted graphs, one containing information related with sensors, such as location, and another one that represents all available navigation routes inside the building.

1. **Hazard Graph:** Contains information related to the location of sensors in the building, and how the hazard can be propagated through the floor. Each node of this graph can be connected, by an edge, to other nodes, representing the cases where the hazard can proliferate directly between them. Each link has an associated weight that is an estimate of the fastest time for the hazard to travel between the two sensors, representing the worst-case scenario. In order to address cases where the spread rate of a fire in a given direction can be different from the spread rate in the opposite direction, the edges of the graph can also be directed. One possible example of this situation is long staircases, where flames can travel upstairs at a higher rate than downstairs.
2. **Navigation Graph:** Represents all possible routes of pedestrian navigation within the building's internal structure. This graph is identical to 'Hazard Graph' since the set of nodes of the 'Navigation Graph' correspond to the same sensor locations represented by the 'Hazard Graph'. The difference is that the weight of the links between the nodes in the 'Navigation Graph' are now estimates of the time taken by evacuees to travel between nodes. The navigation graph also has fewer edges than the hazard graph, since a fire can travel from one node to another, but this transition may not be walkable by a user. Taking into account situations where the time necessary to travel between nodes can diverge according to the direction, similarly to the 'Hazard Graph', the edges on the 'Navigation Graph' can also be directed. For example, fleeing upstairs takes longer than fleeing downstairs.

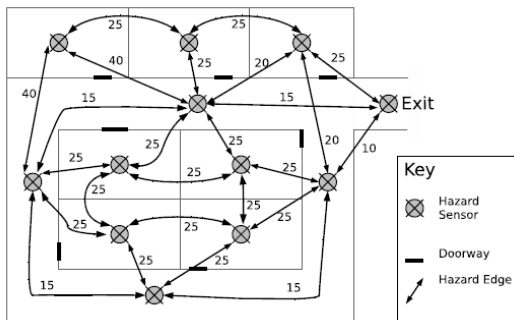


Figure 3.7: Hazard graph

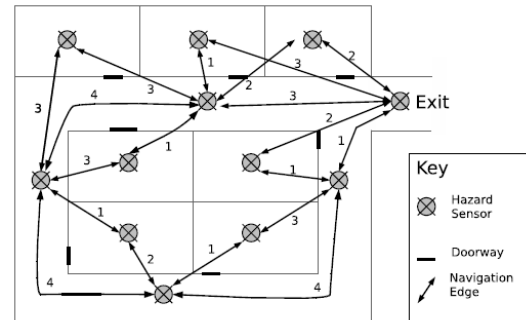


Figure 3.8: Navigation graph

3.3.2 Hazard Time Computation

In the presented solution it is necessary to predict the spread of the hazard for each node. In other words, each node must predict at which time it will be hazardous. The ‘hazard time’ for x node (Hx) is given by the following formula:

$$H_x = \min_{p \in P, p = \langle p_1, p_2, \dots, p_n = x \rangle} \left(\sum_{i=1}^{n-1} F_{p_i, p_{i+1}} \right)$$

where P is the set of all possible paths beginning in one node of the network and ending in x , and $F_{p_i, p_{i+1}}$ is the amount of time it takes for a hazard to spread from p_i to p_{i+1} . This allows the use of a breadth-first search algorithm to trivially calculate the hazard times.

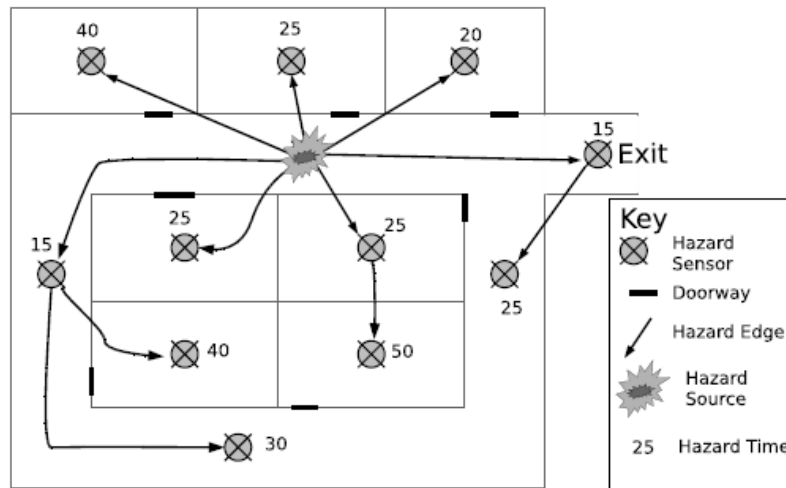


Figure 3.9: Calculated hazard times [3]

3.3.3 Safest path calculation

In order to calculate the safest path to exit the building, the presented approach argues that the safety of a path to an emergency exit node is the minimum time difference between the hazard reaching each node on that route and an evacuee starting at the beginning of the route reaching these nodes. In other words, it is the window of time an evacuee has to reach each node before the node becomes dangerous.

Formally, given that the safety of an empty path on an emergency exit is equal to the time it takes for the hazard to reach that specific exit, we have:

$$S(< v_{\text{exit}} >) = F_{v_{\text{exit}}}$$

So for a given route p where:

$$p = \langle v_i, v_{i+1}, \dots, v_{\text{exit}} \rangle$$

And adding a new vertex at the beginning, results in the route:

$$p' = \langle v_{i-1}, v_i, v_{i+1}, \dots, v_{\text{exit}} \rangle$$

The proposed formula for the calculation of the safety of the path is:

$$S(p') = \min_{S(p) - R_{v_{i-1}, v_i}, H_{v_{i-1}}}$$

Where R_{v_{i-1}, v_i} represents the time for a human to travel from v_{i-1} to v_i .

The logic behind the first term of the minimum is that any delay beginning on an evacuation path makes the path less safe by the amount of the delay and that a human is delayed walking to a path, before starting on it and behind the second term is that no matter how safe an adjacent path is, the hazard might still reach an evacuee first.

These formulas are implemented by both the navigation and hazard algorithms when happening an emergency, so that both algorithms for the propagation of the hazard times and the safety times run simultaneously. This means that they constantly update the times it takes to encounter danger and the times it takes to reach safety whenever we get new information. At each location (node), it keeps a record of which nearby locations can be used as escape routes and how secure those routes are.

3.3.4 Results

In conclusion, in the presented solution, both algorithms, hazard propagation and safe path, need to run simultaneously when an emergency occurs, being continuously updated as new information is received. Every node stores the identifiers of all adjacent nodes to which it can lead evacuees, and stores the safety of each node's exit path.

This approach has the ability to handle arbitrary graphs and by modeling hazard and evacuee speeds through the dangerous environment, using an algorithm that does not use static data, but adapts to the dynamics of the scenario in order to find the exit path.

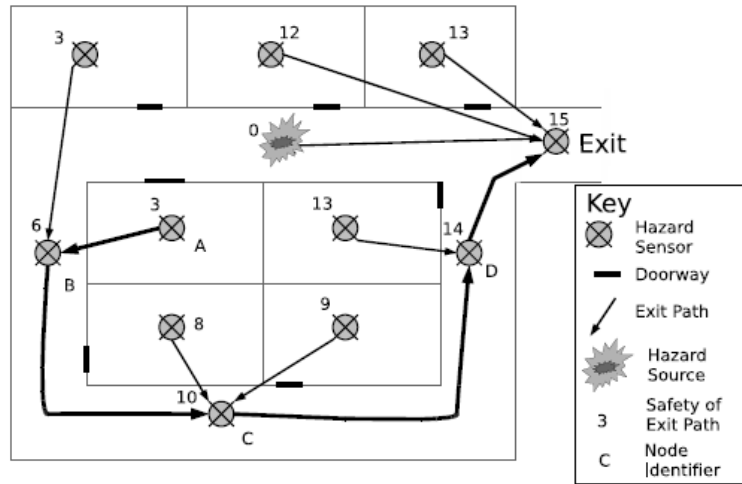


Figure 3.10: Example safety path

3.4 IoT-Based Intelligent Modeling of Smart Home Environment for Fire Prevention and Safety

The approach proposed by Faisal Saeed, Anand Paul, Abdul Rehman, Won Hwa Hong and Hyuncheol Seo from the Department of Computer Science and Engineering, Kyungpook National University, in 2018 [20], describes a wireless sensor network using multiple sensors for early detection of house fires based on sensors that collect data, transmitting them to a processing unit using the ZigBee protocol, and communicating an emergency to the user using GSM technology.

3.4.1 ZigBee Communication Protocol

In this paper, the authors chose the ZigBee, a low-power wireless communication protocol designed for short-range, low-data-rate and low-cost wireless networking, for establishing communication between the sensors and the Processing Unit.

The ZigBee protocol has some key features and characteristics:

1. Low-power consumption: Designed to operate on battery power for long periods.
2. Mesh networking: ZigBee devices can form a mesh network, with several devices communicating with each other, allowing to extend the network's range.
3. Low data rate: Optimized for applications that require low data transmission rates, such as IOT applications, typically ranging from a few kilobits per second up to 250 kbps.
4. Reliability and Security: ZigBee has mechanisms for reliable data delivery and recovery from device failures, also providing security features to protect data transmission and

maintain network integrity, with mechanisms of encryption, authentication, and access control.

3.4.2 System Overview

As we can see in figure 3.11, the proposed fire detection system architecture is constituted by four major parts:

1. **Sensor Nodes:** Smoke, gas, and heat sensors were deployed in different areas of the smart home in order to monitor each area and communicate the data collected to the processing unit, using the ZigBee protocol. Thresholds were defined for each sensor, and when a threshold is reached, the alert is sent to the Processing Unit.
2. **Processing Unit (Home Sink):** Receives information from the sensors and the decision of fire detection is taken based on that information from the sensors but also from the user's response. When a sensor node sends an alert, the sink automatically sends a message to the user, using GSM communication, and based on the user's response or alert notification from the other sensors, the sink takes a decision and generates an alarm in case of fire. The system is also capable of sharing the event information with the emergency service units and the surrounding houses to alert them about the current situation. Raspberry PI (RPI), was used as the primary processing unit due to its popularity and versatile functionalities.
3. **GSM Communication System:** Due to the very high cost of a false alarm, in order to mitigate this problem, a GSM module that sends a pre-warning alert to the user's phone was implemented, allowing the user to confirm the emergency and avoid false alarms.
4. **Alarm System:** Common alarm system present in homes that emits sound signals when an emergency occurs.

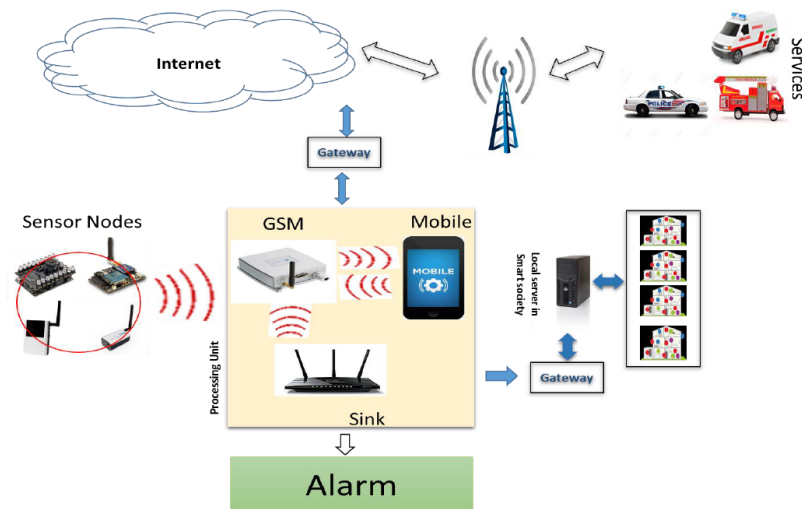


Figure 3.11: Overview of System Architecture

3.4.3 Algorithm

For the proposed system, the algorithm that decides if an emergency is occurring is based on thresholds for the different sensors in the different rooms, meaning that, for example, the thresholds for smoke, gas and temperature can be higher in the kitchen than in the bedroom, which makes sense due to the different characteristics of the rooms in the house and the activities carried out in them. Sensors are always measuring the conditions in the area, and when a threshold is achieved, an alert is emitted to the Processing Unit. Before triggering the emergency alarm, the Processing Unit sends an Short Message Service (SMS) to the user through the GSM Module and waits for a confirmation of the emergency from the user, or waits to receive an alert from another sensor. In summary, and as described in figure 3.12, there are two ways for the algorithm to consider that a fire is occurring:

1. Two or more sensor values exceed the sensors' threshold values.
2. One sensor reaches the defined threshold, sending an alert by SMS and the user response contains a fire confirmation message.

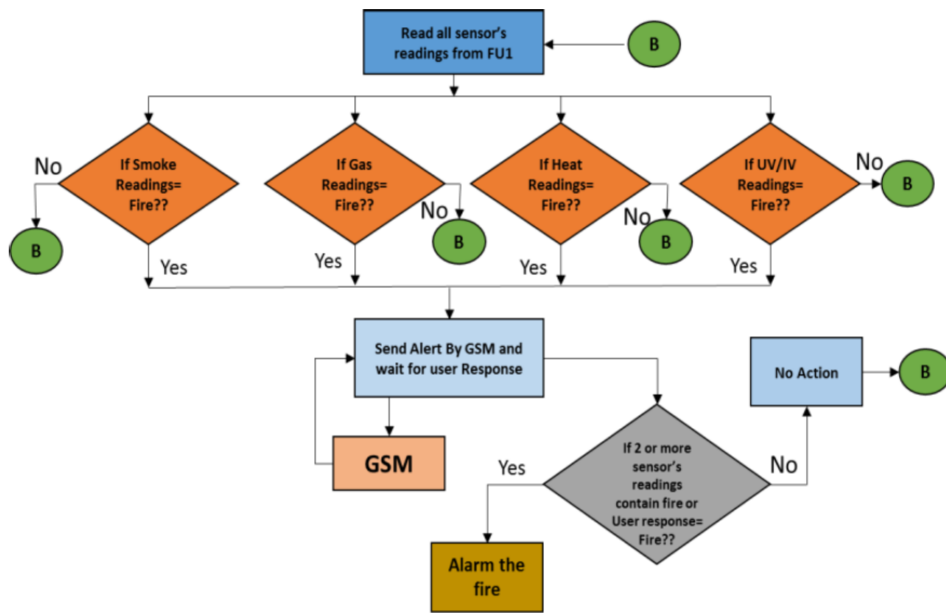


Figure 3.12: System Decision Diagram

3.4.4 Conclusion

This paper proposed a system for fire detection in houses, to correct two major flaws of the currently used systems: the use of a single sensor for event detection and false alarms generated. In order to achieve his goal, smoke, gas, and heat sensors were used, and a GSM Module was implemented, allowing the user to confirm the emergency and to reduce the number of false

alarms originated by the sensors when a value exceeded the defined threshold but no fire was occurring. The presented experiences showed that the system was efficient and accurate, with the authors testing the system many times and producing almost zero percent of false alarms.

This work contributed more to the part of emergency detection, since no study was carried out on the evacuation part, as it was not designed for that purpose, but to be installed in common houses, that does not require a complex evacuation plan as happens in large buildings.

3.5 Summary

In this chapter, it was explored four important research works that supported the development of this thesis, allowing us to understand which are the most common approaches and their advantages and limitations when building an evacuation system. This subsection will serve as a summary and comparison between the different proposed solutions.

Of the group of solutions explored, three of them explored evacuation algorithms, while in [20] the focus was on incident detection and how to avoid false alarms, but also providing important information about wireless communication protocols, such as ZigBee, which can be very useful in systems of this nature.

Analyzing the solutions that worked on evacuation, all of them used graphs to represent the building's navigable network. In the solution [3], two graphs were used, one to represent the navigation paths and another to the sensor network. In the majority of the solutions, the nodes represent locations where users can be present and the edges represent walkable paths between two locations.

The metrics for the safest path calculation are reflected in these graphs through the weight that is assigned to each edge, and in the explored works there are different approaches. In the solution [3], the authors used a time-based formula, where the safety of a link (i, j) is given by the minimum of the time necessary for a user walk from i to j and the hazard spreading from i to j . In [11], the authors suggest that the safety of a link (i, j) is determined by multiplying the distance between i and j by the hazard intensity associated with that particular connection. In solution [16], authors implemented an LCDT-Based method, where the weight of an edge is influenced by the physical length(L), the capacity(C) and density(D) of users, and the trustiness(T). This last parameter is a value between 0 and 1 representing the trustiness of the segment, defined with the help of the information provided by sensors.

Related to the used sensors, smoke and temperature are the most common, allowing to define thresholds that indicate when a hazard is occurring. The solution [16] complemented these two sensors with a camera, allowing to understand the density of users in a certain area, which is an improvement as it allows the system to avoid guiding users along highly congested paths, increasing the time required for evacuation and consequently its effectiveness.

Analyzing the system architecture, papers [3] and [11] propose a distributed system archi-

tecture in order to achieve a robust system in emergency scenarios, where at any time certain components can fail. In the solution [16], the building is divided into different areas that have a responsible node, but this node still needs to use a central node that contains information about the entire building in order to have an overview and choose the safest path, which represents a single point of failure.

Considering all the information presented, this thesis aims to create a system inspired by these existing works, with the goal of addressing and surpassing some of the limitations mentioned earlier.

Chapter 4

System Design

In the previous chapters, some of the approaches present in the literature related to the development of evacuation systems for large buildings were analyzed, exploring important topics that must be considered in the process of developing the proposed dynamic evacuation system.

This chapter will serve to document, analyze and justify the main decisions made in both the overall system structure. Will be presented the system proposed in this thesis, starting with a brief description of it, and then explore the architecture of the system and all its components in detail, in order to provide a rigorous explanation of the system and how all its components interconnect with each other.

4.1 System Overview

The main goal of this work was to develop a distributed system where the building is composed by different areas that have a machine responsible for monitoring the environmental conditions of the area, with the help of information received from sensors, that are deployed throughout the building.

Each area will be able to communicate with neighboring areas, to send summarized information about what is going on in its area and also receive this information from other areas. This way, each area will have information about all the building and will be able to calculate the best path to reach an exit. The information indicating the right path to follow can be conveyed to users through devices that will be installed in the common areas, such as corridors and leisure spaces, allowing that when leaving private spaces, they immediately obtain information on the path to follow.

The proposed system will consist of a server and three different nodes, Decision Nodes, Sensor Nodes and Guidance Nodes. The Decision Nodes will be responsible for computing the safest path of a certain area using a shortest path algorithm, which will be applied to a graph that will

contain all the paths that users can use in the building. The weight of each link in the graph will be directly linked to its security, varying depending on how much the incident affects it. Decision Nodes will also exchange summary information between them in order to obtain an overview of the building, to do this, they will run an algorithm that will determine when new information should be sent and also process the information received.

The Sensors Node will be responsible for monitoring the conditions of each part of the building and transmitting this information to the Decision Nodes, so that it can be processed and reflected in the navigation graph.

Guidance Nodes are devices installed throughout the building that will communicate to users the right direction to follow in order to exit the building safely. This information will be calculated by Decision Nodes that will transmit it to Guidance Nodes.

A configuration server will be available for initial building configurations, such as importing the internal floor plan and entering all general building information.

In the following subsections, it will be explored the system architecture, details about each component and the developed algorithms that will be used in this solution.

4.2 System Architecture

As described in section 2, a distributed system is, at first sight, the most reliable solution, although this type of architecture is more complex and requires more development time compared to a centralized system. In order to obtain a robust system, able to operate when no connection to the exterior is available, a hybrid system is proposed, with the building divided in different areas, with the goal of not having a single point of failure but a system that is still able to operate when certain components are affected by the hazard. Each area will represent a certain part of the building, for example a floor, and a machine will be responsible for monitoring the ambient conditions of this part of the building and communicating summary information to neighboring areas.

To obtain this goal each area will have 3 different nodes (Sensor, Decision and Guidance Nodes), explained in detail later, that will allow to make decisions, indicate the right path to follow to the users, monitor building conditions like temperature or smoke and share summary information between areas.

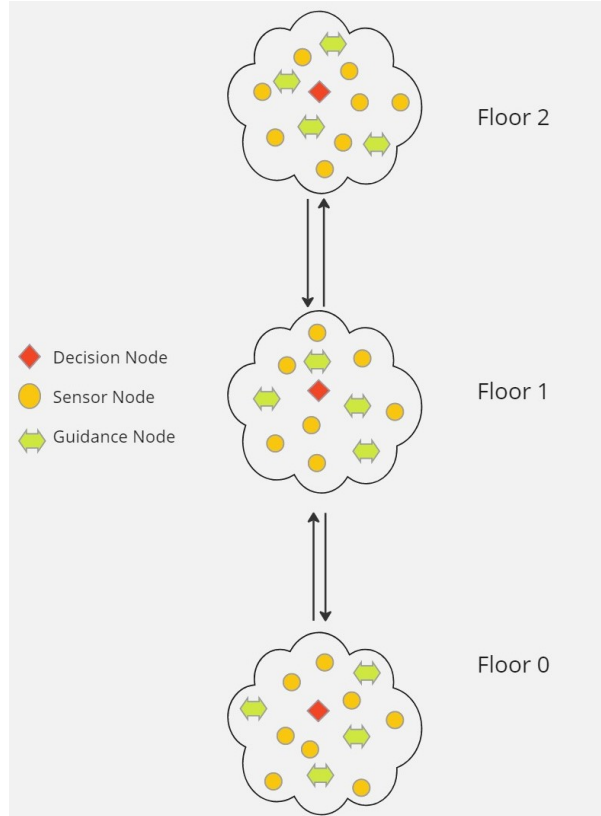


Figure 4.1: Building division by areas

As explained above, the proposed system is composed essentially by 4 components, as shown in Figure 4.2:

1. **Configuration Server:** Supported by an API and a Database, receives, in a first moment, the building plant and general information, processing this data to extract all the necessary information for, in the future, constructing the navigation graph. This server will be used by the **Decision Nodes** to obtain information about its area and always has updated information about the building configuration.
2. **Sensor Nodes:** Responsible for measuring ambient values such as temperature and smoke and transmitting this information to the correspondent Decision Node.
3. **Decision Nodes:** Responsible for processing real-time information from a certain area, using information received from Sensor Nodes, and reflecting this information in the navigation graph, which will be used to calculate the safest evacuation path in this area. When a decision is made, it will communicated to the Guidance Nodes.
4. **Guidance Nodes:** Must receive information from **Decision Nodes** and indicate the direction to users evacuate the building safely.

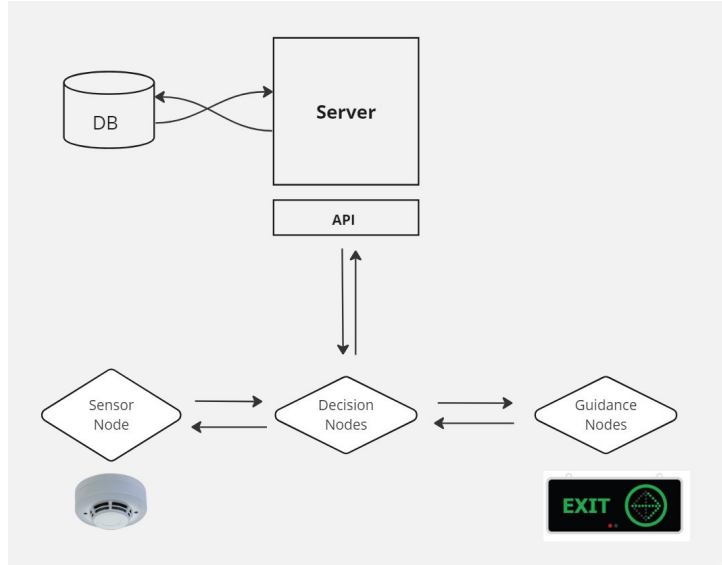


Figure 4.2: Proposed System Architecture

4.2.1 Building Representation

One of the most important characteristics of this work was the use of an indoor mapping format standard to represent the internal characteristics of the buildings. As explored in section 2.1, two main indoor mapping formats were considered for this work: Apple’s IMDF and IndoorGML.

Considering the requirements, in order to provide the essential information for the system to work, we need a format that allows us to represent public spaces, like corridors and public rooms and the correspondent navigation paths, but also represents the location of Sensor Nodes and relate to the areas to be monitored.

As the time needed to represent large buildings was one of the most important criteria in the choice, it was decided to use IndoorGML as the indoor mapping format. Since there are open-source tools for this format, it is simpler to use and meets all the requirements, allowing the definition of an indoor navigation network of the building and the definition of the different areas through the use of description tags.

Each area of the building must be represented in the IndoorGML file, with information related to the available navigation paths, shown in 4.3, in order for the system to process this data and construct a computational representation of the building where operations to find the safest path to reach an exit will be performed.

An important aspect when constructing the navigation path is that the location of each node, let’s call it ”Navigation Node”, will correspond in reality to one ”Guidance Node” present in the building. Therefore, during this report ”Guidance Node” and ”Navigation Node” will be concepts that are related, since when we are constructing the navigation path, we are planning where Decision Node will be deployed in the building.



Figure 4.3: Example of IndoorGML Navigation Paths

In order for the backend server to distinguish which nodes represent building exits when processing the IndoorGML file, the different types of nodes must have the following description tags:

1. Navigation Node area: "Area:name_area".
2. Navigation Nodes that are exits: "type:exit".
3. Sensor Nodes: "type:sensor".

The system also needs to know the location of each Sensor Node and how the values read by them affect each navigation node. In order to have this information, each area must also have a representation of the Sensor layer, as shown in figure 4.4



Figure 4.4: Example of sensor layer in IndoorGML

In the Sensor Layer, each node represents the location of the Sensor Node and, using Inter-Layer Connections, each navigation node must have only a responsible sensor, but each sensor can be responsible for controlling several Navigation Nodes.

4.2.2 Configuration Server

The configuration server component will provide the ability to set initial configurations of the building where the evacuation system will be installed, but also where the IndoorGML file, containing the building's indoor characteristics, can be imported and processed. This component must include the following functionalities, making them available through an Application Programming Interface (API):

1. Insert New Building: Ability to receive general information about new buildings, such as name, length, width, city, and country.
2. Import IndoorGML File: Associate an IndoorGML file to an existing building, processing the file in order to obtain the different areas of the building and the nodes and edges that will be used to construct the navigation graph from this area. It must also, obtain the location of the sensors and the respective zone that they control.
3. Navigation Graph Information: It must be capable of responding to requests from Decision Nodes when they need information about the navigation graph of their area, responding with the nodes and edges from the requested area.
4. Decision Node Information: Decision Nodes from a certain area can request IP Address information from neighboring Decision Nodes in order to communicate with them; this server must provide this information.

Regarding IndoorGML file processing, the configuration server needs to extract the following information from the MultiLayeredGraph element:

- Nodes: Coordinates, type of nodes (navigation or sensor), if the node is an exit, correspondent area and metadata.
- Transitions: How the nodes interconnect between them, coordinates from the origin and the destination, calculate the physical distance in meters, and identify whether it is a connection between different areas or within the same area. The distance between two nodes is calculated through the application of the Euclidean distance formula, where points $A(x_1, y_1, z_1)$ and $B(x_2, y_2, z_2)$ have a navigable link between them:

$$AB = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

- Sensors: Extract the location of each sensor and the nodes that are monitored by them.

4.2.3 Decision Nodes

Decision nodes are the most important components of the system. A decision node is responsible for monitoring a certain area, processing information received from sensors and calculating the safest path to exit the building, whether directly to an exit in its area or guiding users to a neighboring area closer to an exit.

As described before, each Decision Node only has detailed information about its area, receiving summary information about neighboring areas and sharing information about its area with them. Each Decision Node will monitor the state of the area with the help of the information provided by the Sensor Nodes, being able to detect which paths are affected, and how much, by the emergency. With this information, Decision Node will be able to calculate the safest path to exit the building from each Guidance Node, transmitting this information to the correspondent Guidance Node that will indicate to the users the direction to follow.

Decision Nodes are permanently running software that can be divided into the following tasks:

- Construct the area navigation graph: Obtain information from the configuration server, nodes and transitions of the area, to construct the navigation graph. Initially, this task must use the configuration server, but later the information must be stored locally in order to be always available this data even when no communication with the server is available.
- Run algorithm for communication between Decision Nodes: It is necessary an algorithm that communicates summary information between Decision Nodes when there is any change in the area's condition. This algorithm calculates the cost of reaching an exit from each Border Node, nodes in the navigation graph that contain transitions to neighboring areas, and transmits this cost to the corresponding neighboring area.
- Run message server: These devices need to be always listening for messages from Sensor Nodes and other Decision Nodes. A listener must be initiated and process every message received in order to extract the information and update the navigation graph. This server will also be responsible for sending messages to other Decision and Guidance Nodes.
- Calculate path to exit: Run the "Path Finding Algorithm" to obtain the safest path from each Guidance Node to the safest exit, needing only to keep information about the next Guidance Node to reach. When the value of the next node to follow changes, then it must be communicated to the correspondent Guidance Node.

Related to the communication with other Decision Nodes, the first time a DN needs to exchange data with another DN, it will need to obtain information about the recipient's IP address. This information can be obtained through the configuration server and must be stored locally for future communications, allowing that in adverse conditions, where it is not possible to communicate with the server, the system can still work correctly and the communication between them can be successful.

4.2.4 Sensor Nodes

Sensor Nodes are devices installed in public areas of the building to monitor variables that can help detect emergency scenarios. They provide information to Decision Nodes that will allow them to calculate the safest path to exit the building, minimizing the distance walked in the most affected areas.

Higher temperatures, the presence of gas or smoke are some aspects indicating that a hazard is occurring. Each Sensor Node will be associated with a certain area of the building and consequently with the Decision Node of that area, and its purpose is to monitor a certain zone of the area and send the information to the corresponding Decision Node, to be interpreted and used in choosing the safest path.

For this work, we consider that the Sensor Node devices can read values of Temperature and Gas/Smoke using, for example, DHT [1] and MQ-2 [21] sensors respectively. For each sensor, the following thresholds are defined in order to assign a Danger Level from 0 to 3 to the value read. The values of reference for the MQ-2 Sensor were estimated with the help of [8], although some experiments may have to be done in each building to obtain more accuracy.

Temperature Interval	Danger Level
$T \leq 30^{\circ}\text{C}$	0
$30^{\circ}\text{C} < T \leq 40^{\circ}\text{C}$	1
$40^{\circ}\text{C} < T \leq 50^{\circ}\text{C}$	2
$T \geq 50^{\circ}\text{C}$	3

Table 4.1: Temperature intervals and correspondent Danger Levels

Gas/Smoke Interval	Danger Level
$T \leq 300$	0
$300 < T \leq 400$	1
$400 < T \leq 500$	2
$T \geq 500$	3

Table 4.2: Gas/Smoke intervals and correspondent Danger Levels

4.2.5 Guidance Nodes

Guidance Nodes are the component of the system responsible for indicating the path to follow in an emergency scenario, guiding the users to the safest existing exit. This component will be installed in corridors and public areas, allowing people to obtain evacuation path information when they exit personal spaces, rooms and other types of private areas.

As stated before, each Navigation Node present in the IndoorGML file will correspond to one Guidance Node deployed in the building. So for example, when we try to find the best path in the Navigation Graph from a given Navigation Node to an exit, it corresponds in reality to the path from the Guidance Node, deployed in the building, to an exit.

The design of the Guidance Nodes is one of the more complex decisions of the system, having to address problems that complicate how efficiently the information is transmitted to the user during an emergency scenario. Problems like the smoke density, which tends to accumulate on the ceiling and does not allow the visualization of signs that are installed in higher zones, but also all the noise that is produced both by the alarms and by the general panic of the users, which makes it difficult to use sound signals.

This topic also has to lead with the problem of not every path is suitable for every person. For example, people with reduced mobility cannot use stairs, so a different approach has to be studied to address this case. Visual and sound signals do not guarantee that every person will understand which path has to follow due to conditions of deafness or color blindness.

Taking all these aspects into account, a list of possibilities was studied, and a final decision regarding the type of indicator to use in Guidance Nodes could be one of them but also a combination of some devices, depending on the different characteristics of each building.

- Sound Indications: These have advantages in scenarios where the smoke hides visual indications, but cannot be so effective due to the strong noise that is common in emergencies.
- Displays: Transmit the information of the safest path to follow using arrows and stop signals that indicate the direction to go. It has the advantage of being easy to understand, but it may represent a complex solution for areas where several paths cross and is not an effective solution for people who have vision problems.
- RGB Tape: Identical to the solution adopted in the aircraft corridor, with the luminous strip indicating the direction of evacuation. It is a more complex solution and may have a high installation cost in large buildings.

4.3 Navigation Graph

The navigation graph represents all the possible paths to follow in the building during an emergency and will have all the information related to the actual condition of the building. Due to the proposed system design, the navigation path will only cover public zones to the users, such as corridors, where the Guidance Nodes will be present, indicating the path to follow. This means that personal rooms or classrooms, for example, will not be covered by this navigation graph by default. However, special cases of rooms with large dimensions, which contain several exits to different corridors of the building, may be covered due to their usefulness in evacuation, allowing crossing these rooms to avoid affected areas.

The information provided by the IndoorGML file, further imported and processed by the server and stored in the database, allows the analysis of the building's navigation network and the construction of a graph structure, composed by a set of vertices and edges, that will allow to apply path-finding algorithms through it. Each Decision Node has to obtain this information

and construct the navigation graph from its designated area, using this graph to calculate the safest path to exit the building in case of emergency. When a Decision Node receives information from Sensors in its own area or from Decision Nodes in other areas, it will process the received data and reflect changes in the graph.

The weight of the transitions from vertices in the graph represents the real distance that has to be traversed to cross this transition. The major difference in this project, compared to traditional solutions, is the fact that each transition will have an associated level of danger(0-3), with level 0 meaning that this transition is in normal condition and level 3 meaning that the transition is seriously affected by the incident. In short, as shown in 4.5, transitions will not have associated a single value of weight, but an array containing the distance that will be covered in each danger level. For example, an array $[0,10,0,0]$ means that 10 meters will be covered in Danger Level 1. This metric will be used to choose evacuation paths with shorter distances traveled in higher levels of danger, to minimize the user's exposure to more affected areas during the evacuation process.

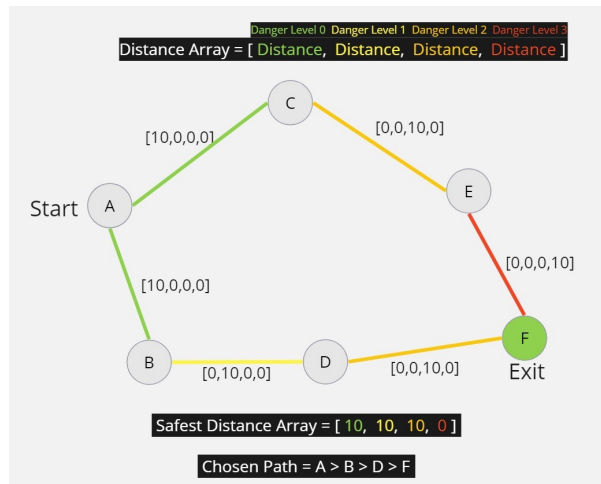


Figure 4.5: Arrays of distances by danger level

When constructing the graph, the decision node will obtain information indicating whether the transition is only within its area or if it corresponds to a transition between areas. In the case of a transition between areas, a node corresponding to the node located in the neighboring area and a transition must be added. Initially, this transition will be created with a very large (infinite) weight, but it will be updated with values received from the Decision Node responsible for the neighboring area. Remembering that the neighboring area transmits the weight of the safest path to evacuate the building from that border node, allowing each Decision Node to have information about what is happening throughout the building, understanding which is the safest path in its neighboring areas, which will allow to choose what is the safest area to follow in order to exit the building. This exchange of information makes the algorithm included in the family of "Distance Vector" algorithms.

Figure 4.6 shows a simplified navigation graph from an Area that has connections with two

other areas. It represents floor 1 and has stairs that allow access to floors 0 and 2. In this graph we have two Border Nodes, A and B, that provide connection to Areas 0 and 2. We can see four nodes that are located in these areas, with transitions initially having a very high (infinite) associated cost. This cost will be updated when Decision Nodes start communicating with each other.

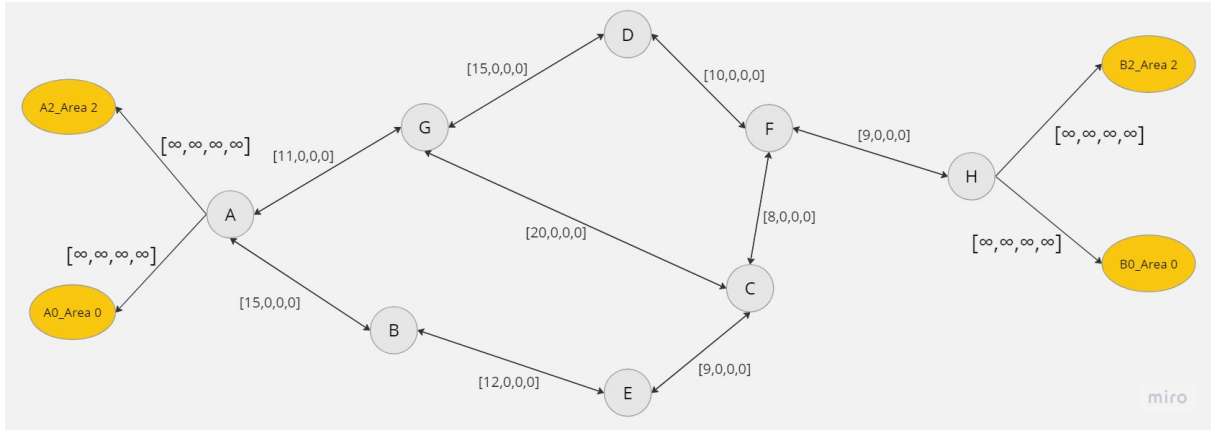


Figure 4.6: Example of Navigation Graph with Neighboring Areas

4.4 Navigation Algorithm

In the navigation graph that each Decision Node will obtain for its area, an algorithm will be performed to find the safest path to exit the building from the location of each Guidance Node. This algorithm will also be used by the Decision Node to calculate the cost of exiting the building from each border node, information that will then be sent to the neighboring areas.

As explored in section 2.3, there are several search algorithms available for safe path-finding problems, such as Dijkstra, Floyd-Warshall, and Bellman-Ford, each with its own set of advantages and disadvantages, suitability for specific scenarios and computational efficiency. As all data regarding the building's navigable network is stored within a Graph object, and the only factors influencing the evacuation path determination are the total distance and the level of danger, opting for the Dijkstra algorithm is a logical decision, since this algorithm can compute the optimal routes from a single starting point to all other nodes within the graph.

As explained, the Dijkstra algorithm requires a source node and, for this work, we will have two variations: one that computes the cost from the source to all other nodes, and another that specifically calculates costs for a given list of destination nodes.

At first sight, the first idea that comes would be to apply the algorithm from each Guidance Node with the set of Exit Nodes as the target, choosing the one that has the safest path. However, this approach would be very inefficient because it requires performing the algorithm for each Guidance Node, which can represent a large number of iterations. A more efficient solution was adopted, where a “dummy” node representing the outside of the building was added to the

graph and connected this node to all Exit Nodes. This way, inverting the Edges of the graph, we can initialize the algorithm from the dummy node, find the best path from the dummy to each node in the graph, and obtain the right path by inverting the result retrieved from the algorithm. With this solution, we can find the best path from each Decision Node to the safest exit performing Dijkstra's Algorithm only one time.

Figures 4.7 and 4.8 illustrate the difference between the two solutions. In the first solution, we need to perform the Dijkstra Algorithm 4 times (with the source node in each of the D, F, C and E Nodes) in order to obtain the best exit, A or B. In the second solution, we consider the “dummy” outside node as the source node and the algorithm gives us the safest path for each node D, F, C and E.

The result returned by the algorithm will be:

Node D: Dummy->A->D (Cost:[15,0,0,0])

Node F: Dummy->A->D->F (Cost:[25,0,0,0])

Node C: Dummy->B->E->C (Cost:[21,0,0,0])

Node E: Dummy->B->E (Cost:[12,0,0,0])

To obtain the actual path from each node to the exit, we only need to remove the dummy and invert the result path, obtaining the following result:

Node D: D->A (Cost:[15,0,0,0])

Node F: F->D->A (Cost:[25,0,0,0])

Node C: C->E->B (Cost:[21,0,0,0])

Node E: E->B (Cost:[12,0,0,0])

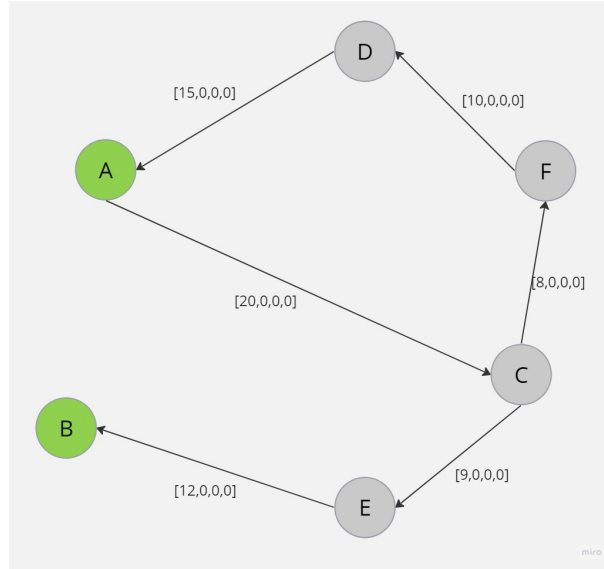


Figure 4.7: Example of Navigation Graph

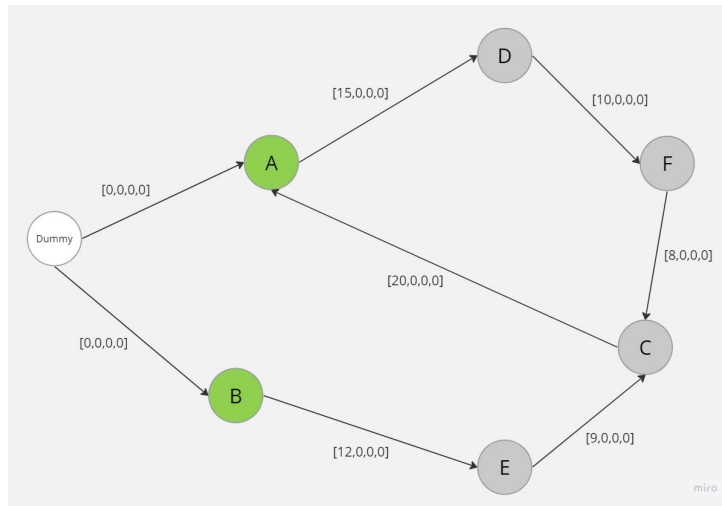


Figure 4.8: Example of Navigation Graph with Dummy Node

4.5 Communication Algorithm Between Decision Nodes

As stated previously, Decision Nodes are responsible for the evacuation process in their area, but do not have detailed information about other areas in the building. Despite this, they need to communicate summary information about their area to each other, so that they can have a global view of the building and guide the users through the safest areas until reaching an exit.

To address this aspect, each decision node must run an algorithm (explained in depth below) that keeps track of the cost of reaching an exit when entering their area by each border node, transmitting this information to the area which the border node is connected to. This way, we

can propagate summarized information from each area to neighboring areas.

The algorithm will be performed every time the system receives new messages with values that change the current navigation graph, which includes changes to the total cost of the path or nodes that make up this path. These messages can be sent by other Decision Nodes, with new summary information, or from Sensor Nodes communicating fresher information about the building's environmental conditions. The algorithm consists of four steps. To facilitate understanding, below we will provide some figures to illustrate each step of the algorithm. In this example, we will use two areas that are connected by two stairs, connecting Node C to Node G and Node F to Node K. The weight of each transition will be a single number in order to simplify the example. The steps are:

- Step 1 – Obtain the area information: Each "Decision Node" will obtain the graph object, containing the navigation graph, a list of exit nodes and a list of border nodes. As stated previously, in the navigation graph provided there are virtual nodes and transitions, with infinite cost, to represent nodes in neighboring areas, as we can see in Figure 4.9.

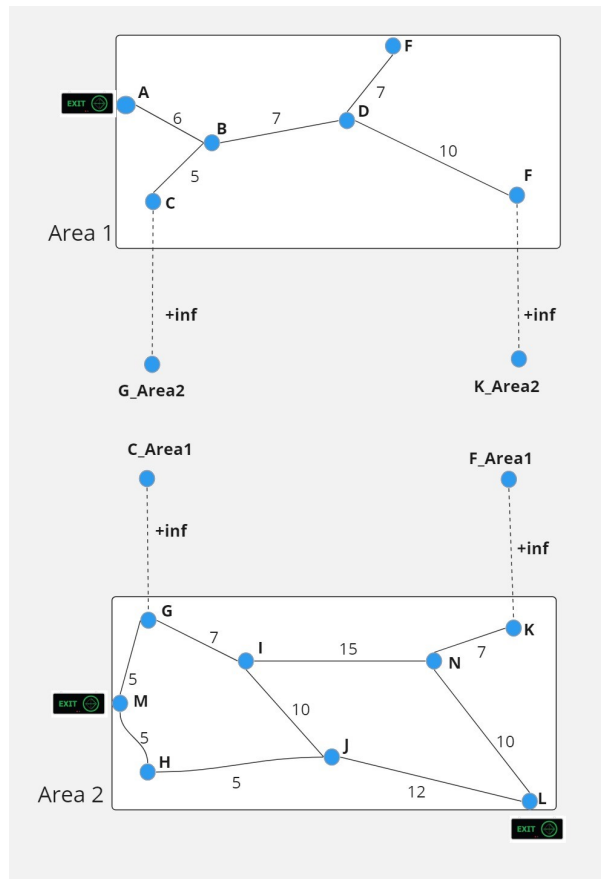


Figure 4.9: Initial Navigation Graph

- Step 2 – Create a structure to store the cost of reaching an exit: Each area will initialize a structure that will have the cost of each combination border node/neighboring area to the safest exit. In this example, Area 1 will add entries for Node C/Area 2 and Node F/Area 2

and Area 1 will add entries for Node G/Area1 and Node K/Area 1. For example, if the border Node “K” had a connection to another area “Area 3”, then another entry would be added to the structure, Node K/Area 3. The cost of each entry will be initialized with an infinite value.

- Step 3 – Calculate cost of reaching an exit - In this step, each Decision Node from the different areas will calculate, for each Border Node_Connected Area combination, the cost of reaching an exit. For each combination, we will ignore the transition that brings the user to the same area. For example, if we are calculating the cost for the combination Node C/Area 2, then the algorithm that finds the safest path ignores the edge that links Node C to the virtual node that represents the Area 2, the Node “G_Area2”. If the cost is different from what is stored in the structure initialized in the previous step, then we update the value and send this information to Area 2, in order for this area to be able to update its navigation graph, setting the value of the transition Node G to “C_Area1” with the new value received, as we can see in Figure 4.10.

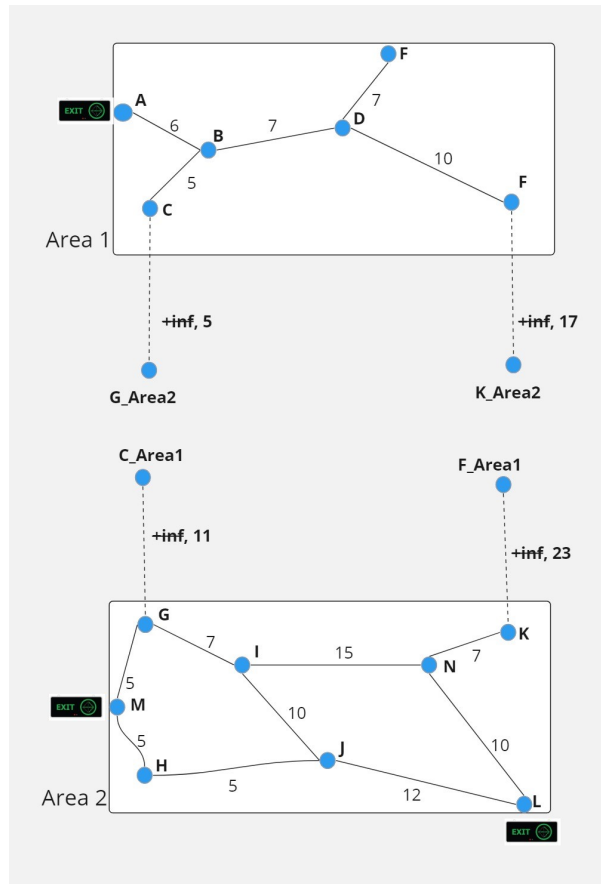


Figure 4.10: Navigation Graph before data exchange between Decision Nodes

- Step 4 – Run the algorithm every time there are changes in the Navigation Graph: Every time that the cost of a transition is updated, the algorithm will run and execute the action described in the previous point. In the example shown in figure 4.11, the value of the

transition Node D to Node F and Node G to Node M changed and then new values were exchanged between Decision Nodes.

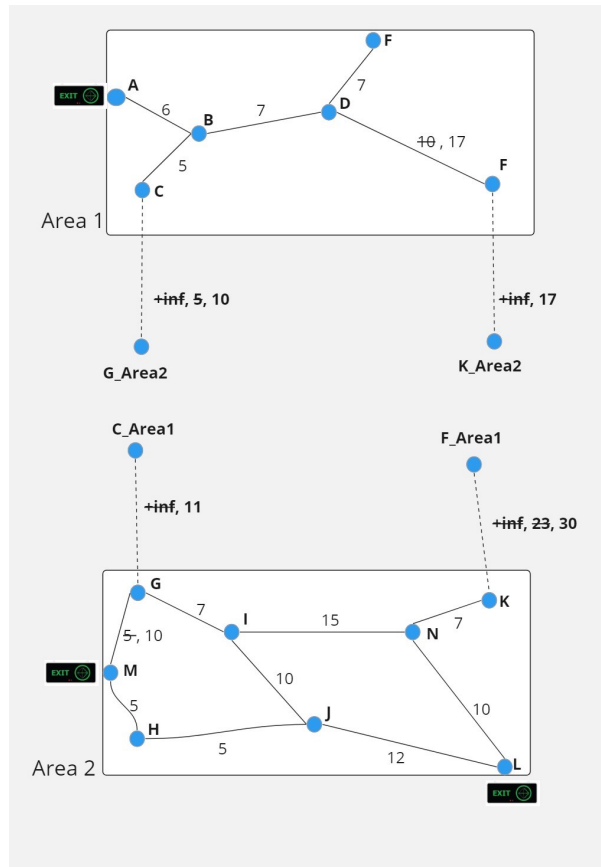


Figure 4.11: Navigation Graph Update

With this algorithm, every Decision Node will know when it needs to communicate summary information with its neighboring areas, allowing them to have a global view of the entire building, which will help in the moment of choosing the safest path that will be transmitted to the users.

4.5.1 Loops Problem in Navigation Graph

During the development of the system, the existence of loops in the navigation graphs was noticed. Figure 4.12, illustrates the problem on the left side, where we have 3 areas that communicate summary information, as explained above, in order to find the cost of reaching an exit through neighboring areas. In this example, we have a loop problem when the Decision Node from "Area 1" needs to calculate the cost of reaching an exit from node E, to communicate this cost to the Decision Node from "Area 0".

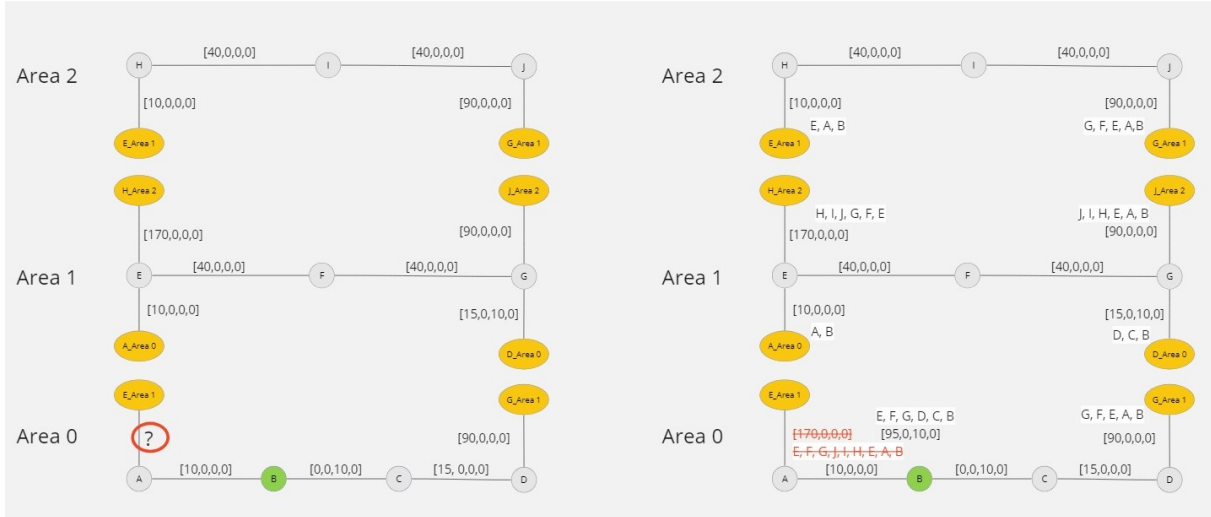


Figure 4.12: Example of loop in navigation graph

When the Decision Node from “Area 1” needs to calculate the cost of reaching an exit from border node “E”, in order to send this information to Decision Node from “Area 0” and ignore the transition from “Node E” to virtual node “A_Area0” as stated before, there are three possibilities available: “D_Area0”, “H_Area2” and “J_Area2”. The obtained cost for these paths will be $[15,0,10,0]$, $[170,0,0,0]$ and $[170,0,0,0]$ respectively. So at first sight, one of the paths that cost $[170,0,0,0]$ will be chosen, as it minimizes the meters traveled at a higher danger level. The problem with these paths is that they bring the user to the “Node E” again, creating a loop.

The solution that we found to solve this problem is composed of two steps. The first one was to store the concatenated path in the virtual nodes, shown on the right side of figure 4.12. For example, node “J_Area2” will store the concatenated path that resulted in cost $[90,0,0,0]$, communicated by the Decision Node of “Area 2”, that will also now send the concatenated path $J \rightarrow I \rightarrow H \rightarrow E \rightarrow A \rightarrow B$ in the message.

The second step is done in the communication algorithm between Decision Nodes, when they are looking for the path with the lowest cost. It will ensure that the concatenated path stored by the virtual nodes does not include the border node that we are looking to find the best path. For example, when the Decision Node from Area 1 is calculating the safest path from border node E, to send to Area 0 (in order to this area update the edge cost from nodes “A” to “E_Area1”) it will receive the following path from Dijkstra’s algorithm with cost $[170,0,0,0]$: $E \rightarrow F \rightarrow G \rightarrow J_Area2$. Then it will check if the virtual node “J_Area2” has node “E” in the concatenated path. If the node “E” is present, then this path will be excluded and the Decision Node of Area 1 will try to find another path. If no other path exists then it will send an empty path with infinite cost to the Decision Node from Area 0.

In the case of image 4.12, the path $E \rightarrow F \rightarrow G \rightarrow D \rightarrow C \rightarrow B$ with cost $[95,0,10,0]$ will be sent by Decision Node from Area 1 to Decision Node from Area 0, communicating the only path that does not create loops.

4.6 Location of Sensor Nodes

Sensor Nodes need to be installed throughout the building, to be able to monitor the conditions both in corridors and in private areas. Emergency sensors are typically mounted on the ceiling or high on the walls, because smoke rises and tends to accumulate near the ceiling before spreading throughout the room. These sensors should not be installed near air vents, as airflow can affect their performance, diluting smoke and delaying detection.

To understand which nodes of the Navigation Graph are monitored by each Sensor Node, in the IndoorGML File construction, as stated in section 2.1, includes a layer of sensor and a layer of navigation which are linked using "Inter-Layer Connection" feature. Although the navigation graph only covers common areas, sensors can be installed in private areas and associated with the closest navigation node.

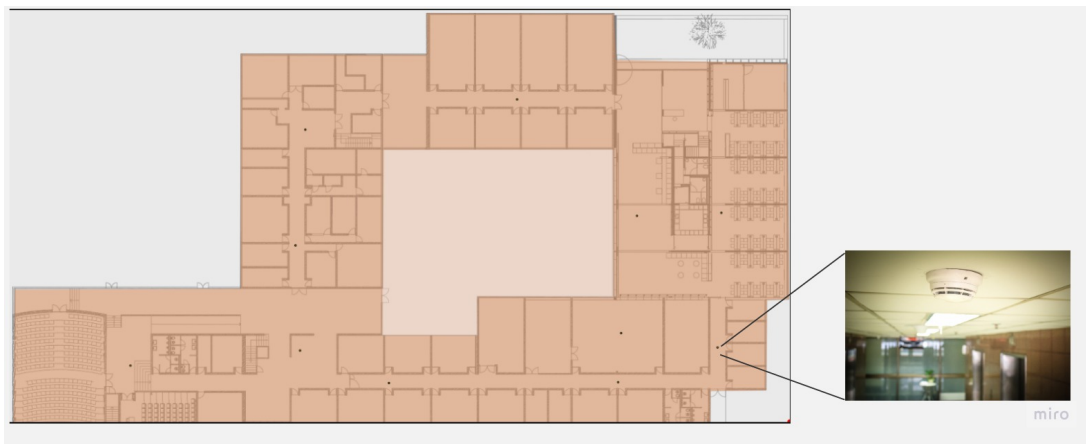


Figure 4.13: Example of Sensor Nodes Location

4.7 Location of Guidance Nodes

As stated before, Guidance Nodes are responsible for guiding the users to the safest exit. Their positioning in the building assumes an important role in order to efficiently convey evacuation information to the users. When exploring the right place to put Sensor Nodes, we understood that smoke tends to accumulate near the ceiling, precluding the idea of installing the Guidance Nodes on the ceiling. On the other hand, devices should not be in a low position that prevents them from being seen in a crowded corridor. This led us to the conclusion that the safest and most efficient position would be placing the Guidance Nodes at $3/4$ of the corridor height. This decision is not fixed, and should vary depending on the characteristics of the building but also on the Guidance Node devices adopted.

Related to the location of each device, when the navigation graph is developed in IndoorGML, the nodes of the graph are commonly placed in locations where the user has to decide the direction to follow, for example at intersections between corridors, but also distributed along

the corridors, so that users always have information available. Taking that into account, it is proposed that Decision Nodes be installed in the location of the nodes of the navigation graph.

4.8 Model and Communication Protocols Between Components

This section describes and analyses the different protocols and requirements to establish communication between the components of the system. This is an important aspect in order to develop a robust system, that works correctly in adverse conditions during a hazard, where we may face several problems of network, that affect how the information is transmitted, forcing us to deal with loss of information in the communication between components. This can represent a simple loss of a value read by a sensor, which will be updated shortly afterward, but also a serious problem if, for example, the safest path is changed during an emergency and cannot be transmitted to the user, leading him/her through the wrong path and putting his/her life in danger.

4.8.1 Decision Node $\leftrightarrow$ Decision Node and Decision Node $\leftrightarrow$ Guidance Nodes

The data exchanged between Decision Nodes and sent to the Guidance Nodes is considered critical, and the system must ensure that all information is delivered with success. If Decision Nodes cannot communicate between them, it means that they will not know the conditions of the rest of the building, not knowing which areas are affected, and whether it is safe to guide the users to them. In the case of the data sent by Decision Nodes to the Guidance Nodes, a lack of communication may lead to problems in guiding the users through unsafe paths, because the safest path can change several times during an emergency, and outdated information may put lives in danger.

Taking this information into account, we need to implement a reliable communication protocol that guarantees that the fresher information is always delivered successfully. At first sight, a protocol based on Transmission Control Protocol (**TCP**) would fill the requirements, providing a reliable and connection-oriented data transmission between devices and ensuring that data packets are delivered accurately and in the correct order, also managing questions of flow control and error recovery. The problem with this approach is that **TCP** will always try to deliver the data, and it could be a problem in our system because older information, that was not delivered successfully is not useful after some time, and **TCP** will always deliver it before deliver fresher information. Furthermore, in case of failure, TCP does not provide to the application information about what has already been delivered and confirmed.

After analyzing all the requirements, we decided to build our solution over User Datagram Protocol (**UDP**), adding an acknowledgment mechanism, with the help of a control value that is sent in the message, and that must be resent by the recipient in order to confirm that the message was received. When a Decision Node needs to communicate with another Decision Node

or with a Guidance Node, it will need to have a structure that keeps track of the messages that must be sent and the corresponding control values. It will periodically analyze the messages it needs to send and will send them until it receives a control message confirming that that message was delivered successfully. If it needs to send a new message, for example, for the area X related to a new cost in a Border Node Y, then it will remove the older one that was in the queue to be sent and will add the new one, to send always the fresher information. This mechanism can be compared to the versioning mechanism from LSA present in OSPF.

4.8.2 Sensor Node → Decision Node

Sensor Nodes need to communicate the sensed values to the Decision Nodes, where they are processed to understand which navigation paths are affected by the hazard.

A message containing the values from each sensor will be sent periodically, with a relatively high frequency, meaning that several messages will be sent per minute. As such, a lightweight communication protocol is necessary. On the other hand, questions like reliability and the guarantee that every message is delivered successfully do not need to be addressed, due to the high frequency. The loss of some messages will not make the system fail because a new message will be sent within a short time.

Taking all these aspects into account, a protocol of communication has been developed, where every x seconds, a message is sent from the Sensor Node to the correspondent Decision Node of the area. This protocol is built over UDP, a fast and lightweight transport layer protocol that provides a connectionless and unreliable communication service, with low overhead and is suitable for real-time applications.

The message has to contain a list of values read by each sensor (smoke, gas) and the ID of the Sensor Node so that the Decision Node can understand which navigation nodes are affected by these values.

```
{
  "type": "sensor_values",
  "data": [
    { "sensorId": "S32", "sensorType": "temperature", "sensorValue": 53.67 },
    { "sensorId": "S32", "sensorType": "gas", "sensorValue": 747.51 },
    { "sensorId": "S32", "sensorType": "smoke", "sensorValue": 724.56 }
  ]
}
```

Listing 4.1: Json structure for Sensor Message

4.9 Database

After exploring the design of the system and all its components, in this subsection we will describe the database requirements.

The Configuration Server must be connected to a Database to develop a robust and efficient system that can recover quickly from a restart and obtain the previously processed data. A MySQL relational database management system was selected for this work, since it is open-source and is relatively easy to set up and manage, also offers a familiar database environment, which allows a higher speed of development. A PostgreSQL relational database was a considered option, since it has all the requirements necessary for this work, but in the end we decided to use MySQL due to my background and due to PostgreSQL being more complex system to set up and maintain. PostgreSQL is more suitable when we need to frequently update data, on the other hand, if we need frequent data readings, MySQL turns out to be the ideal choice. Taking this into account and analyzing our system, there will not be frequent data updates, once the initial configuration of the system is done, most of the operations in the database will be queries, which led us to conclude that MySQL represented the right choice.

Figure 4.14 shows the proposed Entity-Relationship (ER) diagram of the implemented database model, containing a set of tables responsible for storing all system data, namely the information imported from the IndoorGML file and all general values about the building's characteristics.

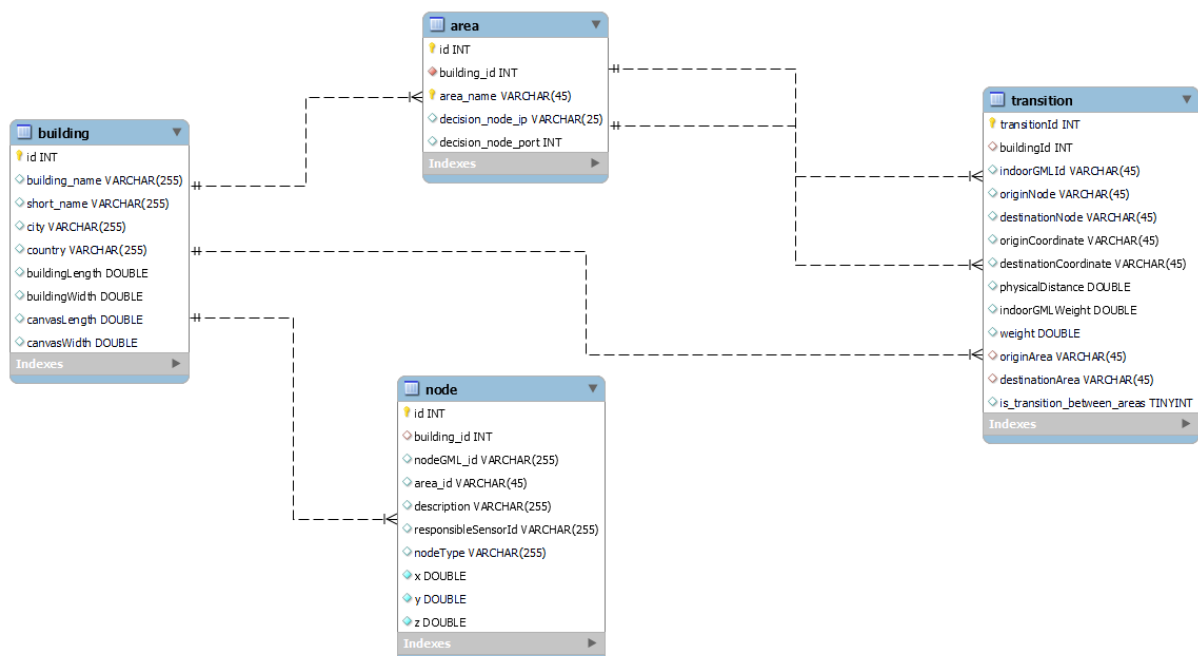


Figure 4.14: ER Diagram

The "building" table will be responsible for storing all the basic information about the

buildings where the system is deployed, described in table 4.3.

The last four fields in table 4.3 were included because the coordinates of each node defined in the IndoorGML file are represented in canvas units. In the canvas frame of reference, the bottom left corner represents the origin point (0, 0) and the top right corner indicates the point (canvasLength, canvasWidth). IndoorGML canvas dimensions have a default value of 400×300 , and when importing an image in InEditor it will always cover this size.

It is necessary to include the building's real dimensions because, during the processing of importing the IndoorGML file, it allows the conversion from the node's canvas coordinates to metric coordinates through the application of a simple proportion, enabling the calculation of real distances in meters between nodes of the navigation graph.

Field Name	Datatype	Description
<i>id</i>	INT	Unique identifier of Building
<i>building_name</i>	VARCHAR(255)	Building name
<i>short_name</i>	VARCHAR(255)	Building name abbreviation
<i>city</i>	VARCHAR(255)	City where the building is located
<i>country</i>	VARCHAR(255)	Country where the building is located
<i>buildingLength</i>	DOUBLE	Building length in meters
<i>buildingWidth</i>	DOUBLE	Building width in meters
<i>canvasLength</i>	DOUBLE	IndoorGML canvas length
<i>canvasWidth</i>	DOUBLE	IndoorGML canvas width

Table 4.3: Building table fields and descriptions

The “area” table will hold information about the different areas from each building, storing variables that will be used for the configuration of the Decision Nodes, like IP and Port values. These values will be requested by Decision Nodes to communicate between them.

Field Name	Datatype	Description
<i>id</i>	INT	Unique identifier of Area
<i>building_id</i>	INT	Associated Building ID
<i>area_name</i>	VARCHAR(45)	Area name
<i>decision_node_ip</i>	VARCHAR(25)	IP of responsible Decision Node
<i>decision_node_port</i>	INT	Port of responsible Decision Node

Table 4.4: Area table fields and descriptions

The “node” and “transition” tables will store information used by Decision Nodes to construct the navigation graph. Table 4.5 shows the fields in table “node”, containing the location of each node, a “description” field that allows identification of nodes that represent exits, “nodeType” to indicate whether a node is a “Navigation Node” or a “Sensor Node”, and the “responsibleSensorId” has the information about a responsible “Sensor Node” from a certain “Navigation Node”, for example, the Navigation Node S20 can be monitored by a Sensor Node “S300”.

Table 4.6 shows the contents of the “transition” table, responsible for storing transition

information. It includes fields that allow us to understand how Navigation Nodes are connected between them, the physical distance of the transition and whether or not it is a transition between different areas.

Field Name	Datatype	Description
<i>id</i>	INT	Unique identifier of Node
<i>building_id</i>	INT	Associated Building ID
<i>nodeGML_id</i>	VARCHAR(255)	IndoorGML node ID (e.g. S20)
<i>area_id</i>	VARCHAR(45)	Associated Area ID
<i>description</i>	VARCHAR(255)	"Exit" if the node represents a exit
<i>responsibleSensorId</i>	VARCHAR(255)	IndoorGML ID of responsible Sensor
<i>nodeType</i>	VARCHAR(255)	"navigation" for navigation nodes and "sensor" for sensors
<i>x</i>	DOUBLE	Location coordinate X (canvas)
<i>y</i>	DOUBLE	Location coordinate Y (canvas)
<i>z</i>	DOUBLE	Location coordinate Z (canvas)

Table 4.5: Node table fields and descriptions

Field Name	Datatype	Description
<i>transitionId</i>	INT	Unique identifier of Transition
<i>buildingId</i>	INT	Associated Building ID
<i>indoorGMLId</i>	VARCHAR(45)	IndoorGML transition ID (e.g. T20)
<i>originNode</i>	VARCHAR(45)	Origin Node IndoorGML ID
<i>destinationNode</i>	VARCHAR(45)	Destination Node IndoorGML ID
<i>originCoordinate</i>	VARCHAR(45)	Origin Node IndoorGML Coordinate
<i>destinationCoordinate</i>	VARCHAR(45)	Destination Node IndoorGML Coordinate
<i>physicalDistance</i>	DOUBLE	Euclidean distance between nodes from the Transition (m)
<i>indoorGMLWeight</i>	DOUBLE	Weight Value provided by IndoorGML File
<i>weight</i>	DOUBLE	Equal to physicalDistance
<i>originArea</i>	VARCHAR(45)	Name of the area where Origin Node belongs
<i>destinationArea</i>	VARCHAR(45)	Name of the area where Destination Node belongs
<i>is_trans_between_areas</i>	TINYINT	"True" if Origin Area is different from Destination Area

Table 4.6: Transition table fields and descriptions

Chapter 5

System Implementation

In the previous chapter, it was explored the proposed system design, describing in detail each component and its function in the system. In this chapter, will explain how the proposed design was implemented, presenting some code snippets and exploring in detail which technologies were used to obtain the final result. It will also discuss the decision-making process that resulted in the adoption of specific technologies and methodologies, highlighting the logic behind these choices.

5.1 System Programming Language

The entire system was developed using .NET, a free, open-source, cross-platform framework, developed by Microsoft, that allows the building of different types of projects, from Web Applications to simple Console Applications.

The C# programming language was used in this project. It can be characterized as an object-oriented programming language designed to be simple, type-safe and efficient. It includes features like garbage collection, type safety, support for modern programming paradigms, such as object-oriented and functional programming, and support for modern software engineering concepts like encapsulation, abstraction, and polymorphism.

An important aspect in choosing this language is the fact that .NET ecosystem provides a large set of libraries and tools for building applications, including the ability to interact with databases, work with files and directories, and create graphical user interfaces.

Another important aspect is the ability to compile the code to different platforms, which is a relevant aspect to obtain a system that can run in Windows but also in Linux distributions, meaning compatibility with the majority of the most used platforms.

The Java language was also considered, but given my background and the comprehensive set of tools that the .NET ecosystem offers, allowing rapid development of a wide range of components, from backend to frontend, the final choice fell on .NET.

5.2 Server

As already mentioned in the System Architecture section of the previous chapter, the Server component will aim not only at processing all the configuration steps related to each building but also at processing the IndoorGML file, providing this information to the Decision Nodes, so that they have all the information about the building and are thus able to construct a computational representation of it. For this component, we decided to use an ASP.NET Core WEB API, a project template that allows the creation of an API capable of processing the different data received and storing it in a database. This stored data will be made available through different endpoints, described below, that can be used by system components.

The implemented API provides the following endpoints:

- **InsertNewBuilding (POST):** Receives general information about the building, such as name, short name, city, country, length, and width. It also receives the dimensions of the IndoorGML canvas, which will allow us, in combination with the provided real dimensions, to obtain the real length in meters of the transitions described in the IndoorGML navigation paths.
- **UploadIndoorGMLFile (POST):** This endpoint is used to import the IndoorGML File associated with a building, receiving as a parameter the “Building ID” and the IndoorGML file. This file will be processed to extract the nodes and transitions that will constitute the navigation graph, but also to identify the different areas of the building. In this process, we also extract the location of each Sensor Node and the associated nodes that are under its responsibility. All this information is stored in the database for further use.
- **GetAllBuilding (GET):** Returns a list with all the buildings in the database.
- **GetNavigationNodesByBuildingArea (GET):** Receives as a parameter the “Building ID” and the “Area ID” and returns the nodes that are associated with this combination.
- **GetNavigationTransitionsByBuildingArea (GET):** Receives as a parameter the “Building ID” and the “Area ID” and returns the transitions that are associated with this combination.
- **GetSensorsListResponsibleNodes (GET):** For a given “Building ID” it returns a list of Sensor Nodes and the correspondent Navigation Nodes that are under his responsibility.
- **GetAreaInformation (GET):** Returns information about a certain area, such as the responsible decision node address. It will be used by other Decision Node to get information to communicate between them.

5.2.1 IndoorGML File Processing

The IndoorGML format is based XML, as mentioned in section 2.1. Therefore, it was necessary to search for libraries and frameworks for reading and parsing XML, so that we could process

important information about the building. With the help of the “System.XML” package, we were able to extract information from the “core:multiLayeredGraph” element from the original GML File, where we can find information about nodes and transitions from the graph provided by IndoorGML. This information will allow the construction of the navigation graph, extracting the coordinates of the nodes and the corresponding transitions between them, but also identifying the areas through the tag “Area: name_area”. In this step, we also extract the location of each Sensor Node and the navigation nodes of the area under his vigilance.

In listing 5.1 we can see how the information about Nodes in MultiLayeredGraph is extracted, getting information such as coordinates, “ID”, and more important which type of node represent. The type of node is extracted from the “Description” field and allows us to identify exits, navigation or sensor nodes and and which area they belong to.

```
public static void Read_Nodes(string filename, int building_id) {

    List<string> AreasList = new List<string>();
    //...
    XmlNodeList xnList =
        xml.SelectNodes(''.../core:MultiLayeredGraph/core:spaceLayers
        /core:spaceLayerMember/core:SpaceLayer/core:nodes/core:stateMember
        /core:State'', nsmgr);

    foreach (XmlNode xn in xnList) {
        Point coordinates_point =
            convertStringToPoint(xn[''core:geometry''].InnerText);

        Node node = new Node {
            building_id = building_id,
            nodeGML_id = xn[''gml:name''].InnerText,
            x = coordinates_point.x,
            //... y and z coordinates };

        string description_raw = xn[''gml:description''].InnerText;
        string[] all_descriptions = description_raw.Split('':');

        for (int i = 0; i < all_descriptions.Length; i++) {
            string description = all_descriptions[i];
            //...
            if (description.Contains(''Area'')) {
                string node_area = //... Extract Node Area

                if (!AreasList.Contains(node_area))
                    AreasList.Add(node_area);

                node.area_id = node_area;
                node.nodeType = ''navigation'';
            }
            if (description == ''type=\\''sensor\\''') //Sensor Node
                node.nodeType = ''sensor'';
        }
    }
}
```

```

        else if (description == ''type=\'\'exit\'\'') //Represents an Exit
            node.description = ''exit'';
    }
}

```

Listing 5.1: Extract Nodes from IndoorGML File

To extract the information about Transitions a similar process is performed, extracting values such as ID, origin and destination nodes, coordinates, weight, but also calculating the physical distance in meters each transition corresponds to, using the coordinates and the real size of the building entered in the settings, as explained above.

At this stage, we need to identify the area to which each transition belongs, or whether it is a transition between areas, storing this information to be used when we construct the navigation graph for each area. With the help of the list of nodes obtained previously, we call a function that searches for the source node and the destination node of each transition, returning the source area and the destination area. This is shown in listing 5.2.

```

//Returns an anonymous type with the origin and destination area
public static object Discover_Transition_Area(Transition transition, List<Node>
    NavigationNodesList) {

    string origin_node = transition.originNode;
    string destination_node = transition.destinationNode;

    string origin_area = NavigationNodesList.Where(node => node.nodeGML_id ==
        origin_node)
        .Select(node => node.area_id).FirstOrDefault();

    string destination_area = NavigationNodesList.Where(node => node.nodeGML_id
        == destination_node)
        .Select(node => node.area_id).FirstOrDefault();

    return new { origin = origin_area, destination = destination_area };
}

```

Listing 5.2: Calculate Transition Real Distance

All this extracted information is stored in the database, in order to be made available to the system components, that can request for this information using the provided API. The connection with the database process is described in the following section.

5.2.2 Connection with database

As planned in the design of the system, a database was created to store information about the buildings of the evacuation system. The Server is the only component that has direct

communication with the database, allowing other components of the system to access this information through the different implemented endpoints of the API explained above.

To establish this communication two packages, “MySQLConnector” and “Dapper”, and several stored procedures were used. We opted for the utilization of stored procedures for several reasons. Firstly, they are precompiled and stored directly on the database server, leading to enhanced performance. Additionally, they reinforce security measures by enabling permissions to be granted solely for the execution of procedures, without revealing the underlying table structure. This, in turn, mitigates the likelihood of Structured Query Language (SQL) injection attacks and unauthorized data access.

One of the most important stored procedures is the “insertNewBuilding”, accessed by the function shown in 5.3. Another one is the operation of getting the Navigation Nodes for a given Building and Area, shown in 5.4. This function receives a “Building ID” and an “Area ID” and returns the list of Navigation Nodes for this set of values.

```
public void insertNewBuilding(Building new_building) {
    using (var connection = new MySqlConnection(Helper.ConnectionString())) {
        connection.Query<Building>('insert_new_building', new
            {new_building.building_name, new_building.short_name,
            new_building.city, new_building.country, new_building.buildingLength,
            new_building.buildingWidth, new_building.canvasLength,
            new_building.canvasWidth}, CommandType:
            CommandType.StoredProcedure);
    }
}
```

Listing 5.3: Connection with Database to Insert New Building

```
public List<Node> getNavigationNodesBuildArea(int building_id, string area_id) {
    using (var connection = new MySqlConnection(Helper.ConnectionString())) {
        return connection.Query<Node>('get_NavigationNodesByBuildingArea',
            new { building_id = building_id, area_id = area_id }, CommandType:
            CommandType.StoredProcedure).ToList();
    }
}
```

Listing 5.4: Connection with Database to get Navigation Nodes for a certain Area

5.3 Sensor Node

The Sensor Node component was developed using a Console Application where we can simulate the values read by each sensor in the building. The development of hardware for sensors does not fall within the scope of this work, due to the cost of implementing a large number of Sensor Nodes and the limited time, so it was decided to choose a solution that allows the selection of

what sensor we want to simulate, and what Danger Level is being read, in order to be obtained a random value in that danger level range, as shown in image 5.1.

```

List of Sensors:
Sensor ID: S100 Area: DCC_Floor2 Monitoring: S79,S88
Sensor ID: S101 Area: DCC_Floor2 Monitoring: S82,S83,S98,S99
Sensor ID: S102 Area: DCC_Floor2 Monitoring: S93,S94
Sensor ID: S103 Area: DCC_Floor2 Monitoring: S92
Sensor ID: S104 Area: DCC_Floor2 Monitoring: S81,S89
Sensor ID: S105 Area: DCC_Floor2 Monitoring: S90,S91
Sensor ID: S106 Area: DCC_Floor2 Monitoring: S80,S97
Sensor ID: S107 Area: DCC_Floor2 Monitoring: S85,S87
Sensor ID: S108 Area: DCC_Floor2 Monitoring: S84,S96
Sensor ID: S109 Area: DCC_Floor2 Monitoring: S95
Sensor ID: S110 Area: DCC_Floor2 Monitoring: S86
Sensor ID: S28 Area: DCC_Floor0 Monitoring: S111,S112,S57,S61,S62
Sensor ID: S30 Area: DCC_Floor0 Monitoring: S13,S19,S20,S9
Sensor ID: S31 Area: DCC_Floor0 Monitoring: S68
Sensor ID: S32 Area: DCC_Floor0 Monitoring: S67
Sensor ID: S34 Area: DCC_Floor0 Monitoring: S25
Sensor ID: S35 Area: DCC_Floor0 Monitoring: S24,S7
Sensor ID: S37 Area: DCC_Floor0 Monitoring: S8
Sensor ID: S38 Area: DCC_Floor0 Monitoring: S22,S27,S4
Sensor ID: S39 Area: DCC_Floor0 Monitoring: S113,S21,S23,S5
Sensor ID: S40 Area: DCC_Floor0 Monitoring: S66
Sensor ID: S69 Area: DCC_Floor1 Monitoring: S43
Sensor ID: S70 Area: DCC_Floor1 Monitoring: S45
Sensor ID: S71 Area: DCC_Floor1 Monitoring: S46,S56
Sensor ID: S72 Area: DCC_Floor1 Monitoring: S41
Sensor ID: S73 Area: DCC_Floor1 Monitoring: S47,S54,S55
Sensor ID: S74 Area: DCC_Floor1 Monitoring: S44,S53
Sensor ID: S75 Area: DCC_Floor1 Monitoring: S51,S78
Sensor ID: S76 Area: DCC_Floor1 Monitoring: S42,S50
Sensor ID: S77 Area: DCC_Floor1 Monitoring: S48,S49

Insert Sensor ID to simulate: S31
Insert Danger Level (0-3): 3
{"type":"sensor_values","data":[{"sensorId":"S31","sensorType":"temperature","sensorValue":89.05}, {"sensorId":"S31","sensorType":"gas","sensorValue":986.6}, {"sensorId":"S31","sensorType":"smoke","sensorValue":564.5}}

```

Figure 5.1: Sensor Node Console Application

The console application shows the "ID" of each sensor and which navigation nodes their cover, and inserting the "ID" and the Danger Level that we want to simulate, the program generates random values in the inserted Danger Level interval, sending a message containing these values, every x seconds, to the corresponding Decision Node of the area. The time between messages is configurable, in order to test different scenarios. The application is built to accept updates in the danger level, for example, a sensor at a certain time may simulate values for a danger level three, but later it may simulate a danger level two and the oldest values will no longer be sent.

```

private static Dictionary<string, int> sensor_values = new Dictionary<string,
    int>();

private static void SendValuesPeriodically() {
    Task.Run(async () => {
        while (true) {
            foreach (var kvp in sensor_values) {
                string area = '';
                for (int i = 0; i < sensors.Count; i++) {
                    if (sensors[i].responsibleSensorId == kvp.Key)
                        area = sensors[i].area_id;
                }
                await SendSensorValues(kvp.Key, area, kvp.Value);
            }
            await Task.Delay(Helper.GetTimeBetweenMessageResend() * 1000);
        }
    });
}

```

```

public static async Task SendSensorValues(string simulated_sensor, string
    area_id, int danger_level) {
    Sensor_Value sensor_data_temperature = new Sensor_Value {
        sensorId = simulated_sensor,
        sensorType = "temperature",
        sensorValue = ... //Get random value in the interval of the desired
            danger level
    };

    //...

    List<Sensor_Value> sensor_data = new List<Sensor_Value> {
        sensor_data_temperature, sensor_data_gas, sensor_data_smoke
    };

    JObject myJson = //Construct json object

    string message = JsonConvert.SerializeObject(myJson);

    //Get Decision Node Area Information
    Area area_info = await
        ServerAccess.GetAreaInformation(Helper.GetSensorNodeBuilding(), area_id);

    Communication_Server.SendUDPMessage(area_info.decision_node_ip,
        area_info.decision_node_port, message);
}

```

Listing 5.5: Function to Sensor Node sends values periodically

5.4 Decision Node

Decision Nodes are the main component of the system, responsible for processing the evacuation route for their area, with the help of information received from Sensor Nodes but also with summary information from neighboring areas exchanged between Decision Nodes. In this section we will explore how Decision Nodes build the navigation graph of their area and how was implemented the algorithm to find the safest path from a certain Guidance Node, the algorithm to exchange summary information between Decision Nodes but also will be described how was implemented the communication protocols with other system components.

At the end, it will be shown how the evacuation process algorithm was developed, understanding how the Decision Node knows the direction to follow from each Guidance Node deployed in the building and how this information is communicated to the Guidance Nodes, so that users can obtain it.

5.4.1 Navigation Graph Construction

The Navigation Graph Construction function returns an “Area_Graph” containing four variables. The first one is a “Graph” structure, that contains a List<Vertex> and a List<Edges>, to represent the vertices and edges of the Navigation Graph. The second and third ones are a List<Vertex> with the exit nodes and a List<Vertex> that contains the border nodes. The last one is a “Dictionary<Vertex, List<Tuple<string,Transition>>>” that, for each border node, has a List containing the name of the area and the correspondent transition that connects the border node to the different neighboring areas.

For example, if the border Node “A” has a connection by stairs to the areas “A1” and “A2”, by transitions “T1” and “T2”, respectively, then the following entry will be added to the dictionary: Vertex: “A”, List<Tuple<“A1”,“T1”>, Tuple<“A2”,“T2”>>. This dictionary is very useful for understanding which areas are connected to the border nodes and through which transition. In listing 5.6 we have a code snippet of the described implementation.

```
private static async Task<Area_Graph> GetNavigationGraph_FromServer() {
    List<Node> navigation_nodes = // Obtain nodes from via Server or saved file
    List<Transition> navigation_transitions = // Obtain via Server or saved file
    //...
    //Add Nodes to graph
    for (int i = 0; i < navigation_nodes.Count; i++) {
        //...
        Vertex vertex = new Vertex { /* Vertex Info */ };
        graph.AddVertex(vertex); //Add Vertex to Graph
        if (navigation_nodes[i].description == "exit")
            exit_nodes.Add(vertex);
    }

    //Add Edges to graph
    for (int i = 0; i < navigation_transitions.Count; i++) {
        Transition transition = navigation_transitions[i];

        if (transition.is_transition_between_areas == false) {
            graph.AddEdge(transition.indoorGMLId,
                graph.GetVertex(transition.originNode),
                graph.GetVertex(transition.destinationNode), transition.weight,
                transition.physicalDistance,
                transition.is_transition_between_areas, 0); //DangerLevel
                Initialized as 0
        }
        else if (transition.is_transition_between_areas == true &&
            !transition.indoorGMLId.Contains("REVERSE")) {
            //Create node representing border node of the neighboring area
            Vertex neighbor_area_vertex = new Vertex { nodeGML_id =
                transition.destinationNode + "'_' +
                transition.destinationArea, area_id = "virtual" };
            graph.AddVertex(neighbor_area_vertex);
        }
    }
}
```

```

        exit_nodes.Add(neighboor_area_vertex); //Add this node to exit nodes
        //DangerLevel Initialized as 0 and with weight of infinite.
        //... Add edges to the Graph
        area_graph.AddNewBoarderNode(transition.originNode);
        area_graph.AddNewConnectAreaToNode(transition.originNode, new
            Tuple<string, Transition>(transition.destinationArea,
            transition));
    }
}

//Add Dummy Node, connected to each exit_node.
Vertex dummy_vertex = new Vertex { nodeGML_id = "dummy", area_id =
    "virtual" };
graph.AddVertex(dummy_vertex);
for (int i = 0; i < exit_nodes.Count; i++) {
    graph.AddEdge("dummy_" + "_" +
        graph.GetVertex(exit_nodes[i].nodeGML_id).nodeGML_id,
        graph.GetVertex("dummy"),
        graph.GetVertex(exit_nodes[i].nodeGML_id), 0, 0, false, 0);
}
//...
return area_graph;
}

```

Listing 5.6: Construction of and Area Navigation Graph

This function allows a Decision Node to obtain the nodes and the edges from its area, using the server or getting locally if they were stored previously. Then, it will construct the “Area_Graph” that must be returned by adding those nodes and edges to the “Graph” object, identifying and storing border nodes and exit nodes in the correspondent variables and using the method “AddNewConnectAreaToNode” from the “Area_Graph” object to construct the “Dictionary<Vertex, List<Tuple<string,Transition>>>” explained before.

The last step in this function is responsible for adding the Dummy Node that must connect to the exit nodes, achieved by creating a new Vertex with “area_id” set as “virtual” and by iterating over each exit node in the list of exit nodes. For each exit node, a new Edge is created that connects this node to the new “dummy” node, with weight and physical distance equal to zero.

Related to the information that is stored locally, currently it is stored data that does not change regularly, where is included the navigation graph of the area and the address of other system components, like neighboring Decision Node or Guidance Nodes. In this phase, when we need to refresh this information we need to delete the files and force the system to get new information from the Server, but in the future a more robust mechanism must be implemented, where we can check periodically if the information that we have locally is updated.

5.4.2 Shortest Path Algorithm

As mentioned in the previous chapter, Dijkstra's Algorithm was chosen to Decision Nodes calculate the safest path to reach an exit from each Guidance Node in the building. The developed algorithm is similar to the common implementation, with the difference of not having a single weight value on the edges of the graph, but an array of doubles, with size equal to four, holding, in each position the distance in meters that must be traveled at the corresponding Danger Levels 0, 1, 2 and 3. The result of the algorithm is the shortest path tree from all Navigation Nodes/Guidance Nodes to the exit minimizing first the distance traveled with the highest danger level, then, if there is a tie, the distance traveled in the next danger level, and so on. We designate this metric as the danger distance metric.

The developed algorithm receives two parameters as input, a Graph and a Vertex, with the graph where the algorithm will be performed and the desired source node, respectively. The execution of this algorithm can be divided into four parts:

1. Initialization: Initialization of three dictionaries, one to store distances from the source vertex, the second one to store the previous vertex of each vertex in the shortest path, useful to return the path from the vertex to the chosen exit and the last one to store the danger level array accumulated from each vertex to the exit. A HashSet containing all vertices in the graph is also initialized. For each vertex in the graph, set the distance to the source vertex to either $[0,0,0,0]$ (if it is the source) or positive infinity and initialize the danger distance array for each vertex with positive infinity values.
2. Main Loop: While there are nodes in the set, we find and remove the vertex with the smallest danger distance in the nodes set. For each neighbor of the current smallest vertex, if it's reachable from the current smallest vertex, we calculate the alternative path's danger distance and total cost and compare the alternative danger distance with the current danger distance of the neighbor. If the alternative path is safer or equally safe and shorter, we update the neighbor's danger distance, total cost, and previous vertex.
3. Result Construction: After the main loop, we construct the final result. First, we initialize dictionaries to store paths, costs, and danger level costs for each destination vertex and then, for each destination vertex in the graph, we traverse the path from the destination to the source using the dictionary that stores the previous node of each node. In the end, we reverse the path to get the correct order from source to destination, round the values in the danger distance array and add the destination vertex, path, cost, and danger distance array to the result dictionaries that will be returned by the function.

A variant of the described algorithm, that receives a third parameter, "List<Vertex> destinations", was also implemented. This variant is used when we only want to know the safest path to a select list of destination nodes, for example, to obtain the path to the exit nodes. The main difference is in the main loop, where we add the visited vertex to a new List of visited

vertex if it is in the destinations list, stopping the loop when the number of vertices in the visited list is equal to the number of vertices in the destination list. This variant also differs in the final step, where we only compute the paths and costs to the selected destination nodes, as shown in listing 5.7.

```
public static DijkstraResult DijkstraShortestPath_DangerLevel(Graph graph,
    Vertex source, List<Vertex> destinations) {
    //...
    var result = new DijkstraResult {
        Paths = new Dictionary<Vertex, List<Vertex>>(),
        Costs = new Dictionary<Vertex, double>(),
        DangerLevelCosts = new Dictionary<Vertex, double[]>()
    };

    foreach (var destination in visitedDestinations) {
        var path = new List<Vertex>();
        var current = destination;
        while (previous.ContainsKey(current)) {
            path.Add(current);
            current = previous[current];
        }
        path.Add(source);
        path.Reverse();

        for (int i = 0; i < dangerDistance[destination].Length; i++) {
            dangerDistance[destination][i] =
                Math.Round(dangerDistance[destination][i], 2); //Round values
        }
        result.Paths.Add(destination, path);
        result.Costs.Add(destination, Math.Round(distances[destination], 2));
        result.DangerLevelCosts.Add(destination, dangerDistance[destination]);
    }
}
```

Listing 5.7: Dijkstra Algorithm Result Construction

5.4.3 Communication Algorithm between Decision Nodes

Related to the communication that has to be performed between Decision Nodes to exchange summary information about the safest evacuation paths, was developed an algorithm, performed every time that occurs changes in the Navigation Graph of his area. This algorithm calculates the path from each border node to the safest exit, sending this information to the correspondent neighboring Decision Node every time this value changes. The following code snippet, in 5.8 represents the developed algorithm.

```
public Dictionary<Tuple<Vertex, Transition>, Exit_Path>
    distance_borderNode_area_exit;
```

```

public async Task Execute() {
    //Calculate the distance from every border node to the safest exit
    foreach (var borderNode in distance_borderNode_area_exit) {
        //...
        //Calculate distance to the different exits
        DijkstraResult result = DijkstraAlgorithm.ExecuteDijkstra(area_graph,
            borderNode.Key.Item1, area_graph.exit_nodes);
        //Choose the best exit
        Vertex best_exit = GetSmallestDangerDistanceArray(result,
            borderNode.Key.Item1.nodeGML_id);

        if (best_exit == null) {
            //There are no available exits. Send +infinite values
        }
        else {
            List<string> concatenated_path = new List<string>();
            foreach (Vertex vertex in result.Paths[best_exit]) {
                concatenated_path.Add(vertex.nodeGML_id);
            }

            Exit_Path new_exit_path = new Exit_Path { ExitNode=best_exit,
                DangerLevelArray = result.DangerLevelCosts[best_exit],
                ExitPath = concatenated_path };

            if (Helper.IsEqualExitPath(new_exit_path,
                distance_borderNode_area_exit[borderNode.Key]) == false) {
                //Identify area and transition to send message
                //Calculate concatenated path to send

                DN_Information data_to_send = new DN_Information {
                    Sender_Area = Helper.GetDecisionNodeArea(),
                    Building = Helper.GetDecisionNodeBuilding(),
                    Area_Transition = transition,
                    concatenated_path = concatenated_path,
                    Cost = result.Costs[best_exit],
                    DangerLevelCosts = result.DangerLevelCosts[best_exit]
                };

                //Add Message to server in order to be sent
            }
        }
    }
    //...
}

```

Listing 5.8: Communication algorithm between Decision Nodes

Exploring more in depth, the algorithm has a “Dictionary<Tuple<Vertex, Transition>, Exit_Path> distance_borderNode_area_exit”, that for each border node stores the cost and the path to reach an exit, and every time that the algorithm is executed, it is iterated over this structure and using the developed Dijkstra algorithm, we calculate the cost of reaching one of the

available exit from each border node, choosing the safest one. If there are no available exits, then we need to communicate to the neighboring Decision Node a cost infinite. Having solutions to exit the area, we compare the obtained path and cost from the algorithm with the previous path cost that is stored in the “distance_borderNode_area_exit” variable, and if these values change, we send a new summary information to the neighboring area update his navigation graph.

5.4.4 Loops Problem in Navigation Graph

In order to implement the mechanism explained in the system design section, where the virtual nodes, that are present in the navigation graph areas, store the concatenated path that is associated with the cost that the Decision Node from the neighboring area has communicated, it was added a field to the Vertex object called “concatenated_path”. The value of this field will be updated when a new message from another Decision Node is received, storing in this way the concatenated path that is the message.

This field will be used to prevent the creation of loops in the navigation graph, because in the Communication Algorithm between Decision Nodes, we ensure the best exit that can be reached from the border node does not create a loop, checking if the border node is not present in the concatenated path of the chosen exit, as shown in the following code 5.9.

```
private Vertex GetSmallestDangerDistanceArray(DijkstraResult result_dijkstra,
string node_id_gml) {

while (result_dijkstra.DangerLevelCosts.Count > 0) {
    Vertex best_exit =
        SmallestDangerDistanceArray(result_dijkstra.DangerLevelCosts);

    if (best_exit.concatenated_path == null ||
        best_exit.concatenated_path.Count == 0) {
        return best_exit;
    }
    else {
        for (int i = 0; i < best_exit.concatenated_path.Count; i++) {
            if (best_exit.concatenated_path[i].Contains('')) {
                // ''Node_AreaName'' Cases
                int index = best_exit.concatenated_path[i].IndexOf('');
                best_exit.concatenated_path[i] =
                    best_exit.concatenated_path[i].Substring(0, index);
            }
        }
        if (best_exit.concatenated_path.Contains(node_id_gml) == false) {
            return best_exit;
        }
        else { //Cycle detected
            result_dijkstra.DangerLevelCosts.Remove(best_exit);
        }
    }
}
```

```

    }
    return null;
}

```

Listing 5.9: Choose of best exit that does not create cycles

The presented function is called in the communication algorithm, to obtain the best exit from a certain border node. This function receives as a parameter the different possible paths for the available exits returned by the Dijkstra Algorithm and the ID of the border node that we are using. Then it chooses the one that has the lowest cost and that does not contain the border node in the concatenated path. For example, if we are calculating the best exit from a border node “A” and one of the available exits contains on the concatenated path field this node “A”, then this exit will be discarded and the function will look for another possible one. If there are no valid exits, then the function will return null and the communication algorithm will send a message to the neighboring Decision Node containing an infinite cost and a null concatenated path.

5.4.5 Communication between system components

Regarding communication between components, it was implemented a communication server that is always running in each component with the goals of receiving and processing messages from other components but also executing the task of sending messages, both the reliable protocol, used to exchange information between Decision nodes and between Decision nodes and Guidance Nodes, but also from the simpler protocol, without message confirmation, used by Sensor Nodes to send the values read to the Decision Nodes.

The communication server is composed of the following functions:

- **Listener:** It is always listening for new messages, identifying what type of message was received, in order to call the right function to extract the information. The implementation will be explained in depth below.
- **Add New Message:** This function is used by the reliable communication protocol and has two different implementations, one for messages to be sent to Decision Nodes and another one for messages that have Guidance Nodes as destination. This function adds the new message to a “ConcurrentBag”, a structure provided by the “System.Collections” package of .NET, that can be characterized as a thread-safe, unordered collection of objects. Two of them have been created to store messages that have Guidance Nodes or Decision Nodes as destinations. Before inserting any value, this function checks whether an older value is already in the collection, updating the value to the new one if necessary. With this mechanism, we ensure that the server always communicates the freshest information.
- **Remove Message:** This function is called by the listener function when a message confirmation is received. This confirmation means that the message has reached the destination

successfully and may be removed from the sending queue.

- **Send Message:** It receives the message to be sent and the destination address and sends it. It's called by the following described function.
- **Check Messages to send:** This function iterates over the “ConcurrentBag” that stores the messages to be sent, messages that can be new or messages for which delivery confirmation has not yet been received, calling the function “Send Message” described above. This process must happen with high frequency, so the value in seconds between a new check was developed in a way that when deploying the system it can be configurable, obtaining the value from editable configuration files. This value must not be so low that will not give time to receive confirmation from the messages sent, but also not too high, making the system react very slowly to messages that have been lost and leading to a poor performance. In the results 6 section, different values will be compared to understand how the system performance is affected.

The “Listener” function, as stated before, is responsible for identifying and processing the messages received. The first step is to recognize the type of message received, by extracting the value present in the field “type” of each message, which can be one of the following:

- “sensor_values”: Message with sensor values.
- “dn_data_exchange”: Message containing summary information about neighboring areas.
- “guidance_confirmation”: Confirmation that a message with a certain control value was delivered to a certain Guidance Node successfully, so it can be removed from the queue.
- “dn_confirmation”: Confirmation that a message with a certain control value was delivered to a certain Decision Node successfully, so it can be removed from the queue.

In the first two types of messages, we need to process the data received to understand how they will affect the navigation graph. In the first one, containing sensor values, represented in 5.10, we extract the values and a danger level is calculated with the help of reference values for smoke/gas and temperature. Then, using a function called “VertexCoveredBySensor”, it will discover which Navigation Nodes/Guidance Nodes are covered by the sensor that sent the values. The adjacent transitions/edges of each node are retrieved by a function called “GetAdjacentEdgesFromNode”, and for each one, we call a function to update the danger level value of the edge.

```
private async Task Listener() {

    // receive message and extract fields type, control string and data

    else if (type == "sensor_values") { //Message containing temperature
        values
        List<Sensor_Value>? data =
            JsonConvert.DeserializeObject<List<Sensor_Value>>(raw_data);
```

```

    for (int i = 0; i < data.Count; i++) {
        if (data[i].sensorType == 'temperature') {
            int dangerLevel =
                DetermineDangerLevelTemperature(data[i].sensorValue);
            List<Vertex> affected_nodes =
                area_graph.graph.VertexCoveredBySensor(data[i].sensorId);

            //Update DangerLevel of adjcent edges from affected nodes
            for (int j = 0; j < affected_nodes.Count; j++) {
                List<Edge> affected_edges =
                    graph.GetAdjacentEdgesFromNode(affected_nodes[j]);
                foreach (Edge edge in affected_edges)
                    graph.UpdateEdgeDangerLevel(data[i].sensorId ,
                        edge.indoorGMLId, dangerLevel);
                //...
            }
        }
        //Similar process for gas/smoke
        await dnalgorithm.New_Execute(); //execute communication
        algorithm between DN if navigation graph has changed
    }
}
//...
}

```

Listing 5.10: Update Edges Weight

An important aspect is that sometimes the same transition can be adjacent to navigation nodes that are covered by different sensors, for example, we can have one node with a danger level equal to two and the other with a danger level equal to three. It was decided that the edges common to these two nodes would have a danger level equal to the highest of the two values, assuming the worst-case scenario. This is a limitation of having the Danger Level associated with the node and not to the edge, and is due to one limitation in the IndoorGML mapping format, where we can only create a relation between nodes in different layers, in this case creating a relation between a Sensor Node in the Sensor Layer and one or more Navigation Nodes in the Navigation Layer. A solution where we can associate a Sensor to one or more Edges would resolve this problem, representing an aspect to investigate in the future.

In the second case, where we receive data from a Decision Node from a neighboring area, we extract the data, where we find information such as the correspondent border node in the neighboring area for which was calculated the path and total cost. Then a function called "UpdateEdgeDangerDistance_Weight" is executed to update the edge in our navigation graph that corresponds to the information received, setting the new value. It will also update the value of the concatenated path in the virtual node with the received path.

Finally, we need to send the confirmation that the message was delivered successfully, so we extract the control value and send the confirmation message to the correspondent Decision Node.

In 5.11 we have a code snippet where we perform the action of receiving the message, extract the necessary values and then send a message to confirm that the message was received successfully.

```
private async Task Listener() {

    // receive message and extract fields type, control string and data

    else if (type == 'dn_data_exchange') { //Used to exchange information
        between decision nodes

        DN_Information data =
            JsonConvert.DeserializeObject<DN_Information>(raw_data);

        graph.UpdateEdgeDangerDistance_Weight(data.Area_Transition.destinNode,
            data.Area_Transition.originNode + '__' +
            data.Area_Transition.originArea, data.DangerLevelCosts,
            data.Cost);

        graph.UpdateEdgeDangerDistance_Weight(data.Area_Transition.originNode
            + '__' + data.Area_Transition.originArea,
            data.Area_Transition.destinationNode, data.DangerLevelCosts,
            data.Cost);

        graph.GetVertex(data.Area_Transition.originNode + '__' +
            data.Area_Transition.originArea).concatenated_path =
            data.concatenated_path;

        //Send a message to confirm

        await dnalgorithm.New_Execute();
    }
    //...
}
```

Listing 5.11: Send confirmation message

5.4.6 Evacuation Process and Communication with Guidance Nodes

In order for each Decision Node to keep track of the path that must be followed by the users, but also be able to communicate this information to the Guidance Nodes, a function that calculates the next Guidance Node to follow was developed. This function is called by the communication algorithm every time a new message is received and we have a change in the navigation graph, storing the information of the next Guidance Node to follow from each Guidance Node in a “Dictionary<Vertex, Vertex>”.

In 5.12 we have a code snippet of the process where we just need to extract the second node from the path, since the first one is the source, and check whether it has changed in relation

to the previous information stored. If the value has changed, then we construct the message to send to the corresponding Guidance Node, adding it to the message server to be sent to the destination.

```
Dictionary<Vertex, Vertex> node_nextNode; //Global variable

private void CalculateNextNodeToFollow() {
    Dictionary<Vertex, Vertex> copy_node_nextNode;
    DijkstraResult result = DijkstraAlgorithm.ExecuteDijkstra(area_graph,
        area_graph.graph.GetVertex('dummy'));

    foreach (var kvp in result.Paths) {
        Vertex origin_node = kvp.Key;
        List<Vertex> path_to_exit = kvp.Value;

        if(path_to_exit.Count >= 2) {
            node_nextNode[origin_node] = path_to_exit[1];
            if (copy_node_nextNode[origin_node] != path_to_exit[1]) {
                Guidance_Data data_to_send = new Guidance_Data {
                    //...
                    Guidance_Node = origin_node.nodeGML_id,
                    Guidance_Node_Next_Node = path_to_exit[1].nodeGML_id
                };
                var myJson = new {
                    type = "guidance_data_exchange",
                    control = control_value,
                    data = data_to_send
                };
                string message = JsonConvert.SerializeObject(myJson);
                server.AddMessage_Guidance(message, ...);
            }
        }
    }
}
```

Listing 5.12: Function to calculate the next Guidance Node to follow

To allow the visualization of the computed evacuation paths, we developed a function that uses the “Dictionary<Vertex, Vertex> node_nextNode” variable and an image of the floor plan of the area to create a representation of which direction must be followed from each Guidance Node. An example of the output is shown in figure 5.3. This image contains the navigation graph, and the color of the edges changes according to the danger level.

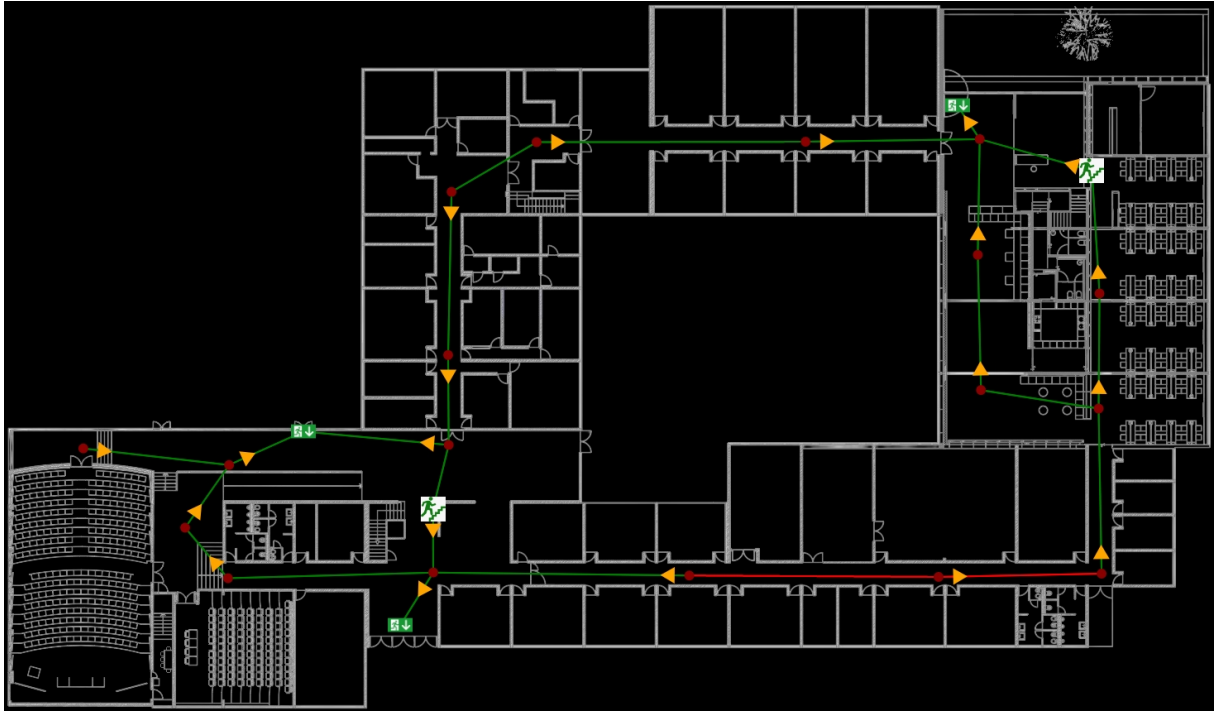


Figure 5.2: Navigation Graph Image

5.5 Guidance Node

Due to the scope of this thesis being focused on software, for the Guidance Node component development, we opted for a program that aggregates all the Guidance Nodes and receives the information from the Decision Nodes. The program developed contains all the communication protocol described in section 4.8, necessary to establish reliable communication between Decision Nodes and Guidance Nodes, and the only difference to a final solution in real world, is the fact that it represents all the Guidance Nodes, showing the information that each Guidance Node in a real scenario would receive individually but in this case in a single application.

The criterion that informed this decision was the fact that it allows faster and simpler testing, a very important factor due to the limited time to develop this work. With this solution, we can test the communication protocols and easily understand which is the next Guidance Node to where each one will point and guide the user.

The solution consists of a console application initializing a “Communication Server”, that will be always listening for new messages from Decision Nodes. When this server receives a message, it will extract the Guidance Node represented and what will be the next Guidance Node to where it will guide the users, saving this information and printing it on the console, as shown in figure 5.3.

```

Guidance Node Hub is running!
Started listening messages

Message received from DCC_Floor0: In the Guidance Node S111 users proceed to Guidance Node S61
Message received from DCC_Floor0: In the Guidance Node S68 users proceed to Guidance Node S67
Message received from DCC_Floor0: In the Guidance Node S67 users proceed to Guidance Node S62
Message received from DCC_Floor0: In the Guidance Node S66 users proceed to Guidance Node S22
Message received from DCC_Floor0: In the Guidance Node S62 users proceed to Guidance Node S112
Message received from DCC_Floor0: In the Guidance Node S57 users proceed to Guidance Node S62
Message received from DCC_Floor0: In the Guidance Node S27 users proceed to Guidance Node S23
Message received from DCC_Floor0: In the Guidance Node S25 users proceed to Guidance Node S111
Message received from DCC_Floor0: In the Guidance Node S24 users proceed to Guidance Node S25
Message received from DCC_Floor0: In the Guidance Node S23 users proceed to Guidance Node S113
Message received from DCC_Floor0: In the Guidance Node S22 users proceed to Guidance Node S4
Message received from DCC_Floor0: In the Guidance Node S21 users proceed to Guidance Node S5
Message received from DCC_Floor0: In the Guidance Node S20 users proceed to Guidance Node S61
Message received from DCC_Floor0: In the Guidance Node S19 users proceed to Guidance Node S20
Message received from DCC_Floor0: In the Guidance Node S13 users proceed to Guidance Node S19
Message received from DCC_Floor0: In the Guidance Node S9 users proceed to Guidance Node S20
Message received from DCC_Floor0: In the Guidance Node S8 users proceed to Guidance Node S23
Message received from DCC_Floor0: In the Guidance Node S7 users proceed to Guidance Node S8
Message received from DCC_Floor0: In the Guidance Node S5 users proceed to Guidance Node S23
Message received from DCC_Floor0: In the Guidance Node S4 users proceed to Guidance Node S27

```

Figure 5.3: Guidance Node Console Application

As explained in section 4.8, this communication needs to be reliable, so our program also extracts the “Control” value, a random identifier, sending this value in the confirmation message that will be delivered to the correspondent Decision Node, confirming that it received the message successfully.

As with the previously described components, information related to Decision Nodes is, in the first case, obtained from the configuration server, but then it is serialized and stored locally, remaining accessible even when no communication with the server is possible. Listing 5.13 shows a code snippet of receiving a message and sending the confirmation.

```

private async Task StartListener() {

    while (!cancellationTokenSource.Token.IsCancellationRequested) {
        UdpReceiveResult receiveResult = await udpClient.ReceiveAsync();
        string data_string = Encoding.ASCII.GetString(receiveResult.Buffer);
        dynamic? jsonObject = JsonConvert.DeserializeObject(data_string);
        string type = jsonObject.type;
        string control2 = jsonObject.control;
        string raw_data = jsonObject.data.ToString();

        if (type == "guidance_data_exchange") { //Used to exchange
                                                information between decision nodes
            Guidance_Data data =
                JsonConvert.DeserializeObject<Guidance_Data>(raw_data);
            Console.WriteLine(@"Message received from {0}: In the Guidance
                               Node {1} users proceed to Guidance Node {2}.",
                               data.Sender_Area, data.Guidance_Node,
                               data.Guidance_Node_Next_Node);

            //Send a message to confirm

```

```
        string response_message = ''ConfirmationMessage:'' + control2;
        var myJson = new {
            type = ''guidance_confirmation'',
            control = control2,
            data = response_message
        };

        string message = JsonConvert.SerializeObject(myJson);
        DataAccess db = new DataAccess();
        Area destination_area_information = await
            db.GetAreaInformation(data.Building, data.Sender_Area);
        bool success = await SendUDPMessage(message,
            destination_area_information.decision_node_ip,
            destination_area_information.decision_node_port));
    }
}
```

Listing 5.13: Guidance Node Message Listener and Confirmation Mechanism

Chapter 6

System Evaluation

This chapter presents the results obtained through the different tests performed with the system. Due to the cost of deploying the entire system in a building, we decided to use simulation tools that allow us to simulate the building and its different areas, simulating several machines that represent the Decision Nodes of the different areas, enabling the distributed calculation of the safest paths, and understand how the system performs in different network conditions.

6.1 DCC Building Deployment

We decided to focus this system evaluation using, as a model, the building Departamento de Ciências de Computadores (Departamento de Ciência de Computadores (**DCC**)), Faculdade de Ciências da Universidade do Porto (Faculdade de Ciências da Universidade do Porto (**FCUP**)), since it is a well known and familiar building, with common characteristics and structural complexity present in the majority buildings of modern cities. This building consists of two floors, with two stairs and one lift to move between them, a large number of classrooms, offices, corridors and large living areas. Figures 6.1 and 6.2 illustrate the two floor plans of the building.

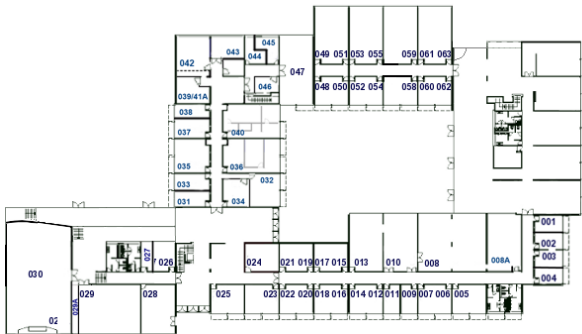


Figure 6.1: DCC Floor 0

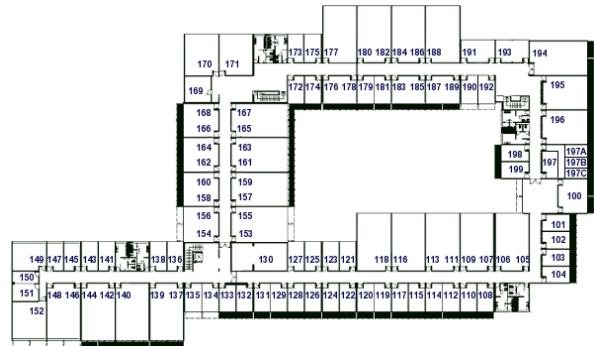


Figure 6.2: DCC Floor 1

We need to take into account some characteristics of this building to understand how the

evacuation system can be handled and how we can plan useful scenarios to test the performance of the system. On the ground floor, we have three exit doors, which are the only exits available in the building, meaning that we always need to reach the ground floor to evacuate the building. We can reach this floor using either of the two stairs or a lift. However, the lift will not be considered since it is dangerous to use in emergency scenarios. All the corridors of the building can be traversed in both directions, meaning that in the building there are no one-way paths.

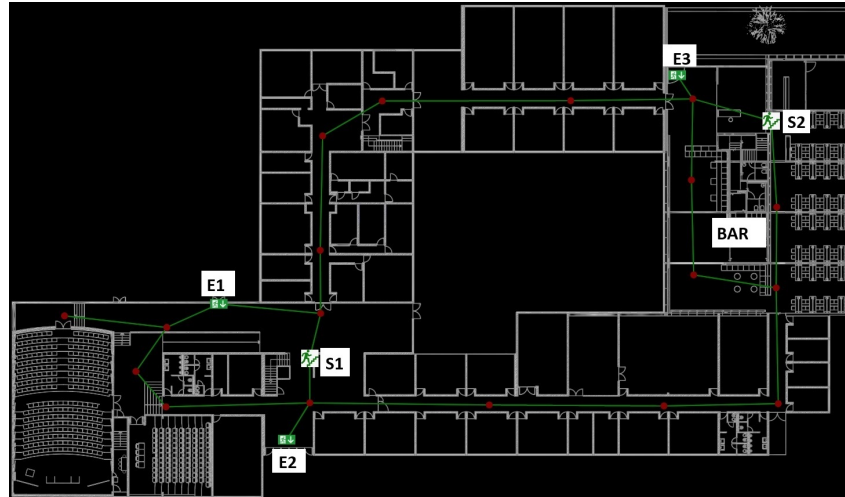


Figure 6.3: DCC Floor 0 Navigation Paths

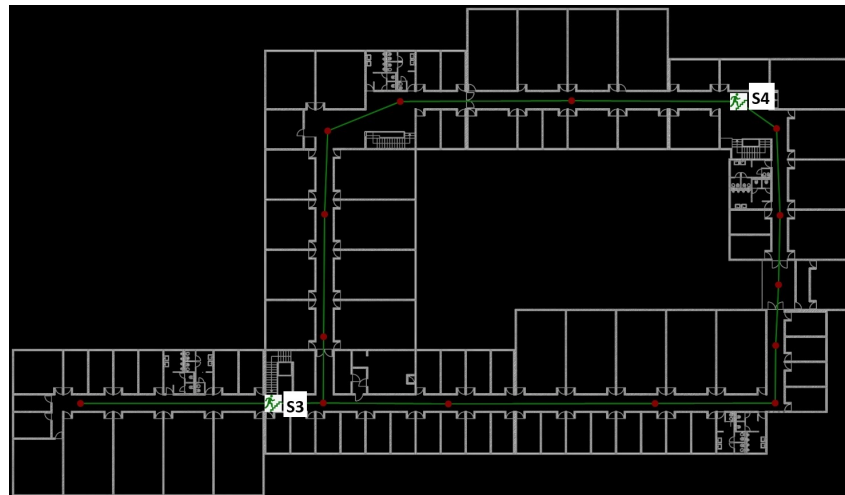


Figure 6.4: DCC Floor 1 Navigation Paths

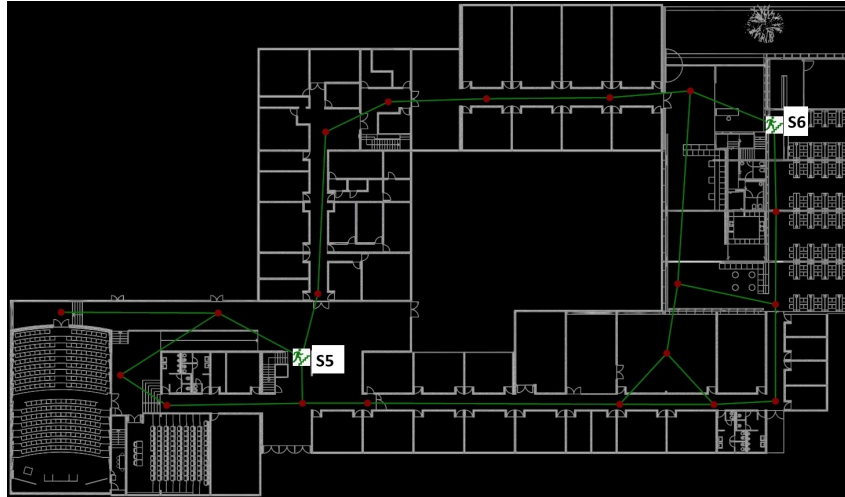


Figure 6.5: DCC Floor 2 Navigation Paths

In order to have a more complete example for testing, we decided to add a third floor that is a copy from the ground, with the difference of having a classroom, shown in figure 6.5, that may be crossed to reach the zone of stairs "S6".

6.2 Test Conditions

One of the most decisive aspects in the evaluation of the system is the ability to have a good performance when the conditions are not perfect, as may occur during an emergency. Taking this into account, the system needs to be evaluated in different network conditions, analyzing whether the communication between components is efficient in different scenarios.

In order to simulate network conditions with communication problems we decided to use the Netem emulator, discussed in , which allowed us to test the system in two different network conditions, described in table 6.1, with varying values of delay and packet loss. This will allow us to understand if the communication protocols and reliable delivery mechanisms work efficiently.

Network Condition	Delay	Packet Loss
Normal	5ms	0%
Weak	1s	50%

Table 6.1: Bandwidth, Delay and Packet Loss values for each Network Condition

Related to the hardware and software used in this tests, it was used Ubuntu 22.02 OS in a machine with an Intel Core i7-10510U 1.8 GHz and 8 GB RAM.

To better visualize and understand the evacuation route provided by the system, it as developed a tool that prints the produced path in an image that represents each floor. This tool firstly print the navigation paths and then in each node it will print an arrow indication which

direction must be followed. In the case where users must go to another area we cannot print the arrow indicating this, therefore, we will use the information present in the application that represents the guidance nodes to understand which direction users should follow in that location.

6.3 Test Cases

Analyzing the building characteristics and how an emergency can affect the evacuation process, we decided to use four emergency cases, described below, to test how the system performs and check whether the users are guided through the safest path.

1. Fire in one exit: In this test case, we will simulate an emergency in the zone of exit "E3" on floor 0. We want to guarantee that the system guides users to the other exits.
2. All exits are affected: Each exit on floor 0 will be affected by a different level of danger, making sure that the system chooses the one that has the lower danger.
3. Go up one floor: In this case, we will simulate that the users are surrounded by fire in the zone near exit "E3". We want to make sure that the system chooses to guide the people to the first floor, using the stairs "S2", and going down again to the ground floor through the stairs "S3", in order to use exit "E2". For users that are near the stairs "S2" it is safer to go up one floor because the distance walked on higher danger levels will be less than reach the exit "E3".
4. Go up two floors: In this case an emergency will be simulated in the ground, as in the previous test case, but also on floor 1, forcing the users that are in the area near stairs "S4" to go to the second floor, go down to the first floor and then to the ground floor, exiting the building using the exit "E2".

We will test the system with the help of these four test cases, changing the network conditions, as described in table 6.1, in order to understand if the system reacts in the correct way even when the network conditions are not ideal.

To have an idea of the standard exit paths when there are no emergencies, the next three images show the shortest path from each Guidance Node to an Exit when no danger is present.

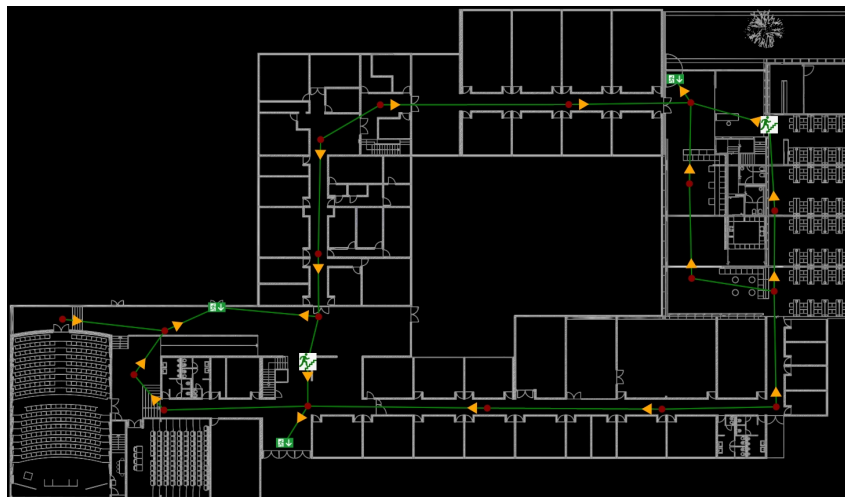


Figure 6.6: DCC Floor 0 Default Exit Paths

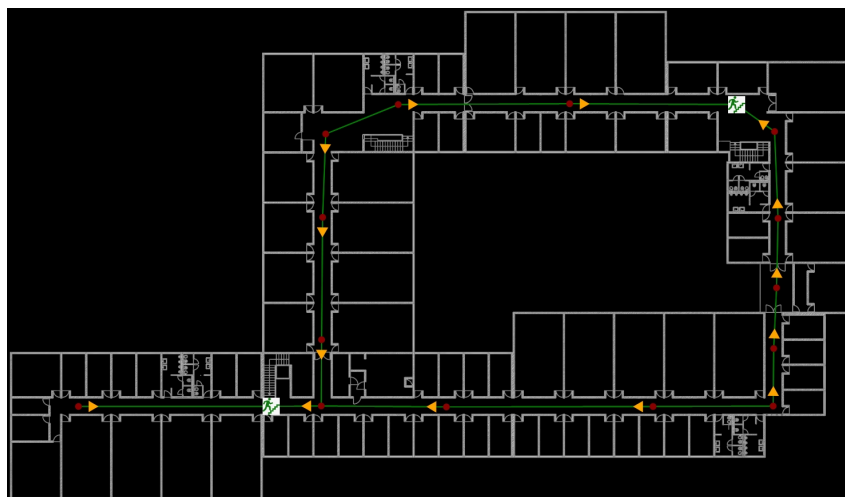


Figure 6.7: DCC Floor 1 Default Exit Paths

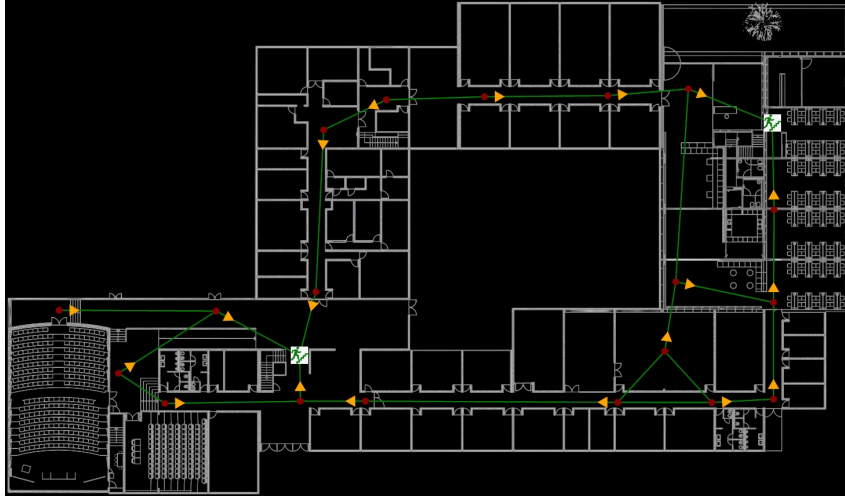


Figure 6.8: DCC Floor 2 Default Exit Paths

6.4 Results

In this section, it will be presented the results obtained in each test case. Our objective is to analyze how the system responds under diverse network conditions during potential hazards. It will be discussed not only the safest path determined by the system but also the behavior of the communication mechanism. We seek to understand if the developed system performs as intended, ensuring the safe and efficient evacuation of building occupants. By testing the performance under various scenarios, we can obtain critical insights into the system's reliability and its ability to adapt to changing conditions.

6.4.1 Test Case 1

In the first test case, we simulated an emergency in the exit zone "E3" with a Danger Level equal to 3, the maximum in our system. Analyzing the results obtained, we observed that with the network in the worst condition, dropping half of the packets, the system was still effective, returning the same results as with the network in excellent conditions. In the next image, we can observe that some retransmissions of packets occurred, which indicates that the developed communication mechanism performed well and was robust to the problems of the network.

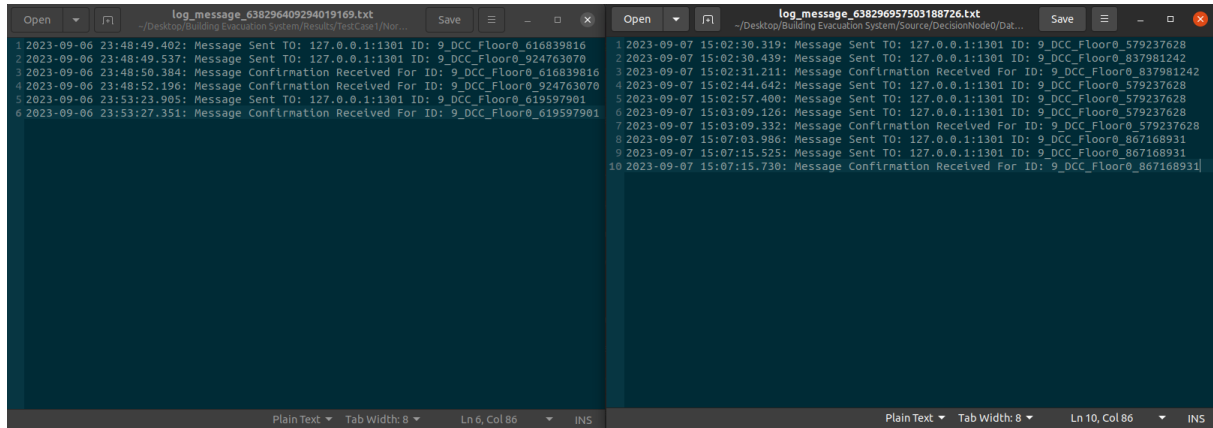


Figure 6.9: Communication on a normal network (left) and on a worst network (right)

On the right side, we can see that, initially, Decision Node 0 sent two messages to Decision Node 1 with summary information, but only received confirmation for the second one. Then, it resents the first message until it gets the confirmation. The same situation occurred when Decision Node 0 had to communicate the emergency on its floor, not receiving the confirmation at first, but only when this information was sent again. In both conditions, the system guided the users to the other two available exits, avoiding the affected exit.

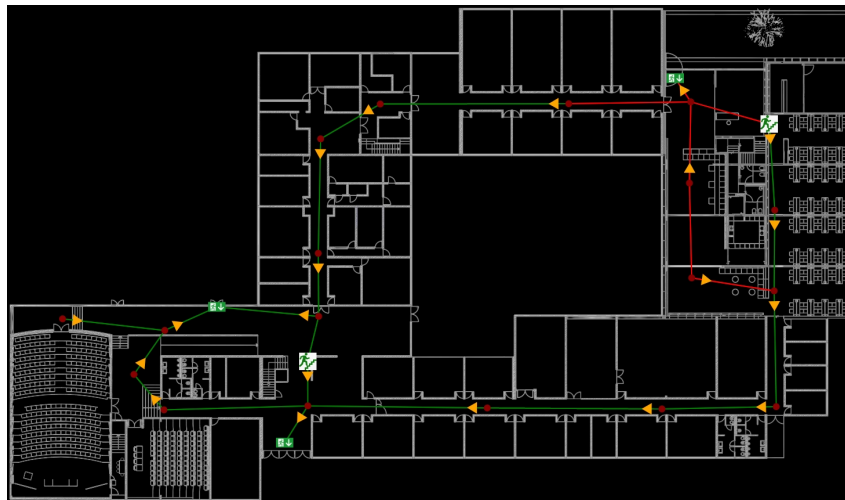


Figure 6.10: Test Case 1: DCC Floor 0 Exit Paths

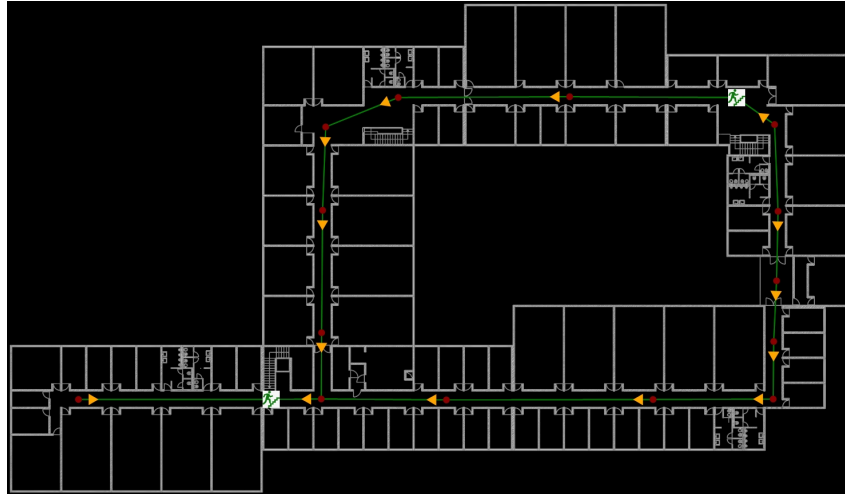


Figure 6.11: Test Case 1: DCC Floor 1 Exit Paths

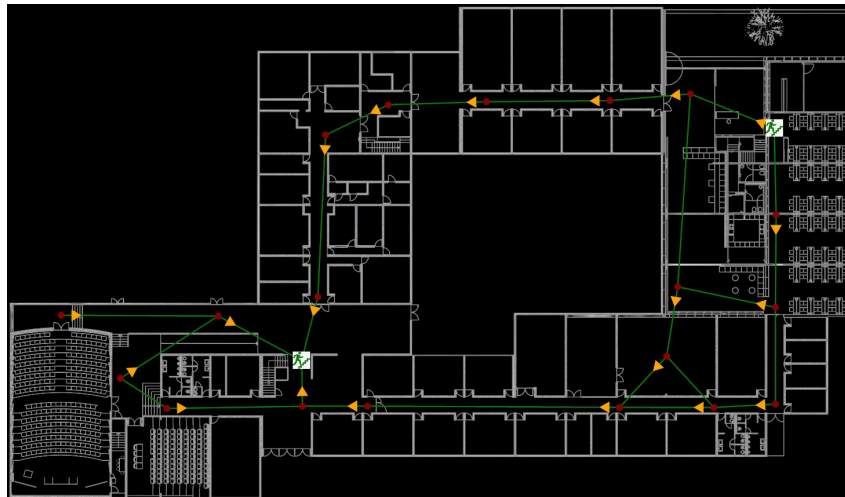


Figure 6.12: Test Case 1: DCC Floor 2 Exit Paths

6.4.2 Test Case 2

In this test, we simulated each exit on floor 0 affected by an emergency, with exits "E1" and "E2" affected by a level 3 danger and exit "E3" by a level 2 danger. With this scenario, we wanted to make sure that the system chooses the exit having the lower danger level, E3, in the top right of the floor image. The system provided the safest exit paths, represented in the following images:

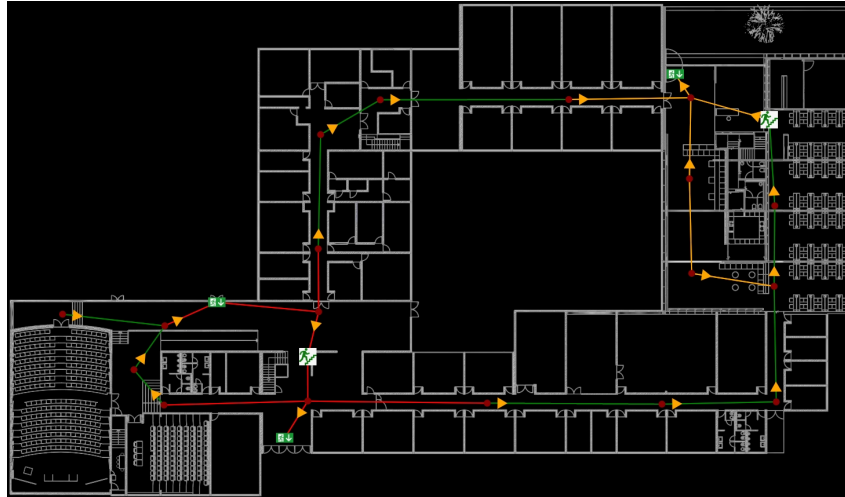


Figure 6.13: Test Case 2: DCC Floor 0 Exit Paths

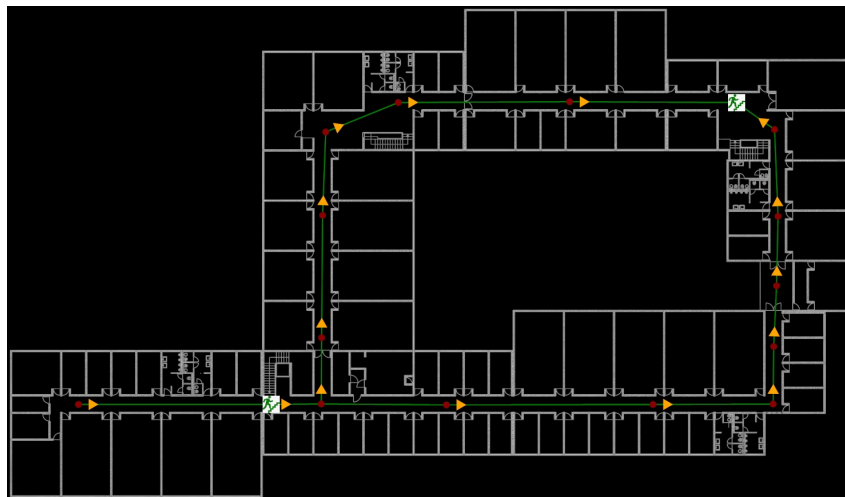


Figure 6.14: Test Case 2: DCC Floor 1 Exit Paths

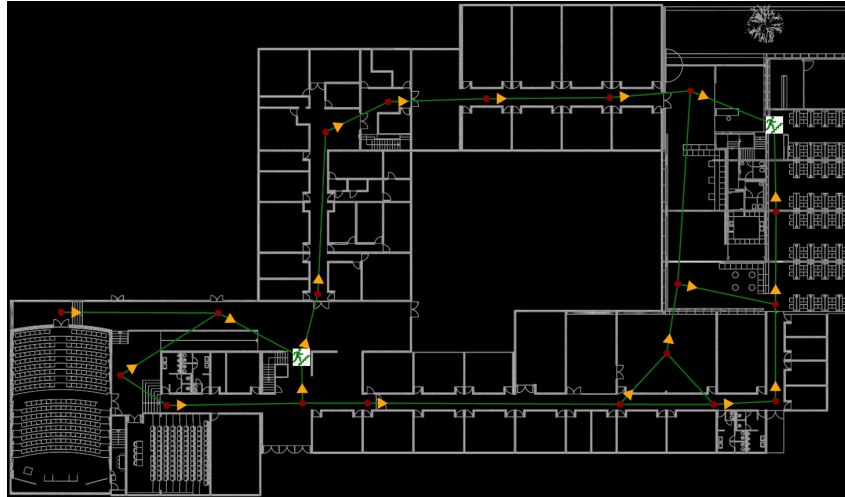


Figure 6.15: Test Case 2: DCC Floor 2 Exit Paths

In this case, the system could not find a completely safe exit, but, as expected, tried to minimize the danger. As we can see in the image 6.13, the users are guided to the exit on the top right of the floor plan. One important aspect is that the users that are in the bottom left zone of the floor plan always have to walk through a path with maximum danger level, with the system choosing the path where this distance is as low as possible. Another curious aspect is that some users will be guided to the first floor, since the distance to reach the stairs is lower than the distance to exit the affected zone. Then, they will be guided to the stairs on the top right of the image and exit the building, returning to the ground floor, but using the exit with danger level two.

In the case where network conditions are poor, we find that the system takes too long to achieve the expected result, requiring more than two minutes to deliver certain messages, which is quite long in an emergency scenario. We can observe this in the message with Control/ID "9_DCC_Floor1_640794109", represented in figure 6.16, where the first message was sent at "23:44:05" and the last one was sent at "23:46:12", receiving confirmation for this message two seconds later.

```

20 2023-09-08 23:41:25.901: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_610055445
21 2023-09-08 23:41:37.841: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_610055445
22 2023-09-08 23:41:49.374: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_610055445
23 2023-09-08 23:42:00.300: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_610055445
24 2023-09-08 23:42:11.014: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_610055445
25 2023-09-08 23:42:21.725: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_610055445
26 2023-09-08 23:42:32.436: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_610055445
27 2023-09-08 23:42:42.946: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_610055445
28 2023-09-08 23:42:45.164: Message Confirmation Received For ID: 9_DCC_Floor1_610055445
29 2023-09-08 23:43:54.075: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1025600338
30 2023-09-08 23:43:54.177: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_341340922
31 2023-09-08 23:44:05.700: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
32 2023-09-08 23:44:05.802: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_605925733
33 2023-09-08 23:44:05.903: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1025600338
34 2023-09-08 23:44:06.003: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_282678699
35 2023-09-08 23:44:17.747: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
36 2023-09-08 23:44:17.848: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_605925733
37 2023-09-08 23:44:17.946: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1025600338
38 2023-09-08 23:44:18.048: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_282678699
39 2023-09-08 23:44:26.148: Message Confirmation Received For ID: 9_DCC_Floor1_1025600338
40 2023-09-08 23:44:29.166: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_282678699
41 2023-09-08 23:44:29.267: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_605925733
42 2023-09-08 23:44:29.370: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
43 2023-09-08 23:44:40.290: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_818650574
44 2023-09-08 23:44:40.392: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_282678699
45 2023-09-08 23:44:40.395: Message Confirmation Received For ID: 9_DCC_Floor1_605925733
46 2023-09-08 23:44:40.494: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_605925733
47 2023-09-08 23:44:40.618: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
48 2023-09-08 23:44:42.617: Message Confirmation Received For ID: 9_DCC_Floor1_282678699
49 2023-09-08 23:44:52.993: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_158566268
50 2023-09-08 23:44:53.092: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_818650574
51 2023-09-08 23:44:53.207: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
52 2023-09-08 23:44:56.629: Message Confirmation Received For ID: 9_DCC_Floor1_818650574
53 2023-09-08 23:45:06.359: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
54 2023-09-08 23:45:06.461: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_158566268
55 2023-09-08 23:45:18.186: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
56 2023-09-08 23:45:18.286: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_158566268
57 2023-09-08 23:45:29.201: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
58 2023-09-08 23:45:29.303: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_158566268
59 2023-09-08 23:45:29.503: Message Confirmation Received For ID: 9_DCC_Floor1_158566268
60 2023-09-08 23:45:40.227: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
61 2023-09-08 23:45:51.154: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
62 2023-09-08 23:46:02.070: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
63 2023-09-08 23:46:12.795: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_640794109
64 2023-09-08 23:46:14.999: Message Confirmation Received For ID: 9_DCC_Floor1_640794109

```

Figure 6.16: Test Case 2: Time to deliver the last message with 10 seconds between message resend

In this test we used a delay of ten seconds for message retransmission, so, to try to improve these results, we adjusted this value to resend messages every five seconds, and, as shown in figure 6.17, the system performance improved a lot, taking less than 1 minute to send all the messages that contain information about the new emergency occurring. We can take as an example the message with ID/Control "9_DCC_Floor1_1029539274", here the first message was sent at "15:09:14" and the last one was sent at "15:10:06", receiving confirmation for this message four seconds later.

```

18 2023-09-09 15:06:19.873: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_1000508758
19 2023-09-09 15:06:19.995: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_409792367
20 2023-09-09 15:06:26.320: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_1000508758
21 2023-09-09 15:06:26.422: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_409792367
22 2023-09-09 15:06:32.544: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_1000508758
23 2023-09-09 15:06:32.648: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_409792367
24 2023-09-09 15:06:38.359: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_1000508758
25 2023-09-09 15:06:38.461: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_409792367
26 2023-09-09 15:06:43.965: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_1000508758
27 2023-09-09 15:06:44.069: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_409792367
28 2023-09-09 15:06:49.372: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_1000508758
29 2023-09-09 15:06:49.475: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_409792367
30 2023-09-09 15:06:51.580: Message Confirmation Received For ID: 9_DCC_Floor1_1000508758
31 2023-09-09 15:06:53.285: Message Confirmation Received For ID: 9_DCC_Floor1_409792367
32 2023-09-09 15:08:25.024: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_83323260
33 2023-09-09 15:08:25.127: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_606470580
34 2023-09-09 15:08:31.659: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_83323260
35 2023-09-09 15:08:31.760: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_606470580
36 2023-09-09 15:08:33.864: Message Confirmation Received For ID: 9_DCC_Floor1_83323260
37 2023-09-09 15:08:38.294: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_606470580
38 2023-09-09 15:08:44.625: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_606470580
39 2023-09-09 15:08:50.553: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_651257277
40 2023-09-09 15:08:50.655: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_69904603
41 2023-09-09 15:08:50.756: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_405413340
42 2023-09-09 15:08:54.803: Message Confirmation Received For ID: 9_DCC_Floor1_405413340
43 2023-09-09 15:08:55.068: Message Confirmation Received For ID: 9_DCC_Floor1_69904603
44 2023-09-09 15:08:57.495: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_651257277
45 2023-09-09 15:08:59.702: Message Confirmation Received For ID: 9_DCC_Floor1_651257277
46 2023-09-09 15:09:14.239: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_619019084
47 2023-09-09 15:09:14.341: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1029539274
48 2023-09-09 15:09:22.545: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_619019084
49 2023-09-09 15:09:22.651: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1029539274
50 2023-09-09 15:09:29.383: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_619019084
51 2023-09-09 15:09:29.485: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1029539274
52 2023-09-09 15:09:35.433: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_619019084
53 2023-09-09 15:09:35.538: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1029539274
54 2023-09-09 15:09:41.057: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_619019084
55 2023-09-09 15:09:41.158: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1029539274
56 2023-09-09 15:09:46.263: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_619019084
57 2023-09-09 15:09:46.365: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1029539274
58 2023-09-09 15:09:51.466: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_619019084
59 2023-09-09 15:09:51.567: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1029539274
60 2023-09-09 15:09:53.672: Message Confirmation Received For ID: 9_DCC_Floor1_619019084
61 2023-09-09 15:09:56.670: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1029539274
62 2023-09-09 15:10:01.776: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1029539274
63 2023-09-09 15:10:06.879: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_1029539274
64 2023-09-09 15:10:10.010: Message Confirmation Received For ID: 9_DCC_Floor1_1029539274

```

Figure 6.17: Test Case 2: Time to deliver the last message with 5 seconds between message resend

6.4.3 Test Case 3

In this test case, the users are surrounded by a hazard on the ground floor and the system must guide them to the first floor, and then down again to the ground floor, using the stairs that bring them to the exit “E2”. This test allowed us to understand whether correct summary information is being exchanged, allowing areas to understand if other areas represent a safer evacuation path.

Both the first and second tests, in normal and weak network conditions, respectively, produced the following evacuation paths:

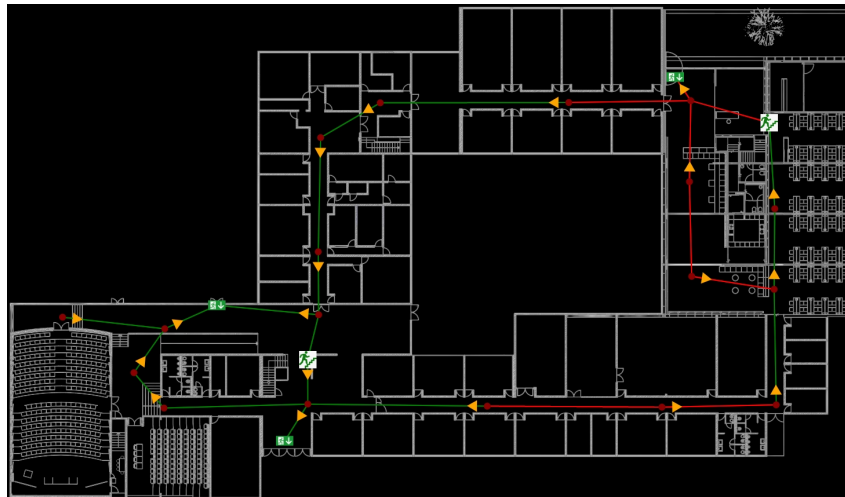


Figure 6.18: Test Case 3: DCC Floor 0 Exit Paths

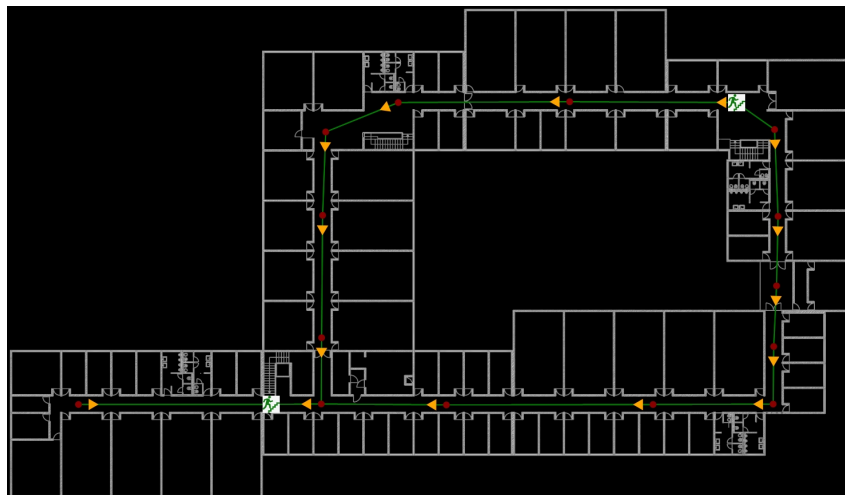


Figure 6.19: Test Case 3: DCC Floor 1 Exit Paths

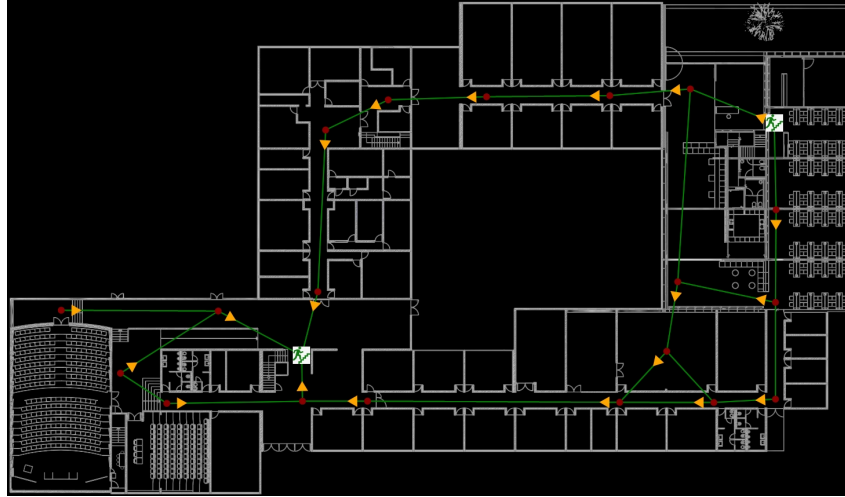


Figure 6.20: Test Case 3: DCC Floor 2 Exit Paths

In figure 6.18, we can see that the users who are surrounded by the hazard are guided to the stairs, leading them to the first floor. In the image, we cannot have an arrow that indicates that the users will use the stairs, so, using the guidance node component that receives the next node to follow from each Guidance Node, figure 6.21, we observe that in the node with ID "S27", the next node to follow by the users will be the "S51_DCC_Floor1", confirming that the system guided the evacuees to the first floor and then to the ground floor.

```

Message received from DCC_Floor2: In the Guidance Node S87 users proceed to Guidance Node S85.
Message received from DCC_Floor2: In the Guidance Node S90 users proceed to Guidance Node S89.
Message received from DCC_Floor0: In the Guidance Node S8 users proceed to Guidance Node S7.
Message received from DCC_Floor0: In the Guidance Node S66 users proceed to Guidance Node S22.
Message received from DCC_Floor2: In the Guidance Node S87 users proceed to Guidance Node S85.
Message received from DCC_Floor0: In the Guidance Node S27 users proceed to Guidance Node S51_DCC_Floor1.
Message received from DCC_Floor2: In the Guidance Node S91 users proceed to Guidance Node S79.
Message received from DCC_Floor2: In the Guidance Node S90 users proceed to Guidance Node S89.

```

Figure 6.21: Next Node to follow by users in Node S27

Analyzing the performance in the two network conditions, we obtained similar results when compared to the previous test case, with several retransmissions, but achieving the desired result in about 1 minute, using a time interval of five seconds between message retransmissions. Figure 6.22 shows some message logs. We may observe that the message that took longer to be successfully delivered took about one minute and ten seconds.

```

8 2023-09-17 17:52:38.479: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_534234838
9 2023-09-17 17:52:40.597: Message Confirmation Received For ID: 9_DCC_Floor1_991734596
10 2023-09-17 17:52:46.045: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
11 2023-09-17 17:52:46.146: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_986013205
12 2023-09-17 17:52:46.248: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_145228224
13 2023-09-17 17:52:53.851: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
14 2023-09-17 17:52:53.953: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_986013205
15 2023-09-17 17:52:54.052: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_145228224
16 2023-09-17 17:53:00.588: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
17 2023-09-17 17:53:00.686: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_986013205
18 2023-09-17 17:53:00.789: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_145228224
19 2023-09-17 17:53:06.707: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_414254750
20 2023-09-17 17:53:06.808: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
21 2023-09-17 17:53:06.909: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_597801142
22 2023-09-17 17:53:07.010: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_145228224
23 2023-09-17 17:53:10.417: Message Confirmation Received For ID: 9_DCC_Floor1_414254750
24 2023-09-17 17:53:10.520: Message Confirmation Received For ID: 9_DCC_Floor1_145228224
25 2023-09-17 17:53:10.638: Message Confirmation Received For ID: 9_DCC_Floor1_597801142
26 2023-09-17 17:53:13.744: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
27 2023-09-17 17:53:20.067: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
28 2023-09-17 17:53:25.780: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
29 2023-09-17 17:53:31.497: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
30 2023-09-17 17:53:37.006: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
31 2023-09-17 17:53:42.310: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
32 2023-09-17 17:53:47.617: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
33 2023-09-17 17:53:52.919: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_830317412
34 2023-09-17 17:53:55.715: Message Confirmation Received For ID: 9_DCC_Floor1_830317412
35 2023-09-17 17:54:28.034: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_608853232
36 2023-09-17 17:54:28.143: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_387717798
37 2023-09-17 17:54:32.927: Message Confirmation Received For ID: 9_DCC_Floor1_387717798
38 2023-09-17 17:54:34.456: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_608853232
39 2023-09-17 17:54:40.174: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_608853232
40 2023-09-17 17:54:42.400: Message Confirmation Received For ID: 9_DCC_Floor1_608853232
41 2023-09-17 17:54:45.683: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_577359707
42 2023-09-17 17:54:51.393: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_577359707
43 2023-09-17 17:54:56.908: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_577359707
44 2023-09-17 17:55:02.010: Message Sent TO: 127.0.0.1:1302 ID: 9_DCC_Floor1_577359707
45 2023-09-17 17:55:04.215: Message Confirmation Received For ID: 9_DCC_Floor1_577359707
46 2023-09-17 17:55:17.113: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_705224693
47 2023-09-17 17:55:22.220: Message Sent TO: 127.0.0.1:1300 ID: 9_DCC_Floor1_705224693
48 2023-09-17 17:55:22.789: Message Confirmation Received For ID: 9_DCC_Floor1_705224693
49 2023-09-17 17:55:24.437: Message Confirmation Received For ID: 9_DCC_Floor1_705224693

```

Figure 6.22: Test Case 3: Message retransmission

6.4.4 Test Case 4

In this test case, we have similar conditions on the ground floor as in the previous test case. However, in this case, floor one is also affected, meaning that evacuees will have to reach floor two and go down to the ground floor again in order to exit the building safely. To have a single safest exit route, floor two will also be affected, leaving only one path completely safe for evacuees to follow. On the ground floor, the hazard will have a maximum danger level, on floor one danger level equal to two and on the third floor danger level equal to one.

Both the first and second tests, in normal and weak network conditions respectively, produced the following evacuation paths:

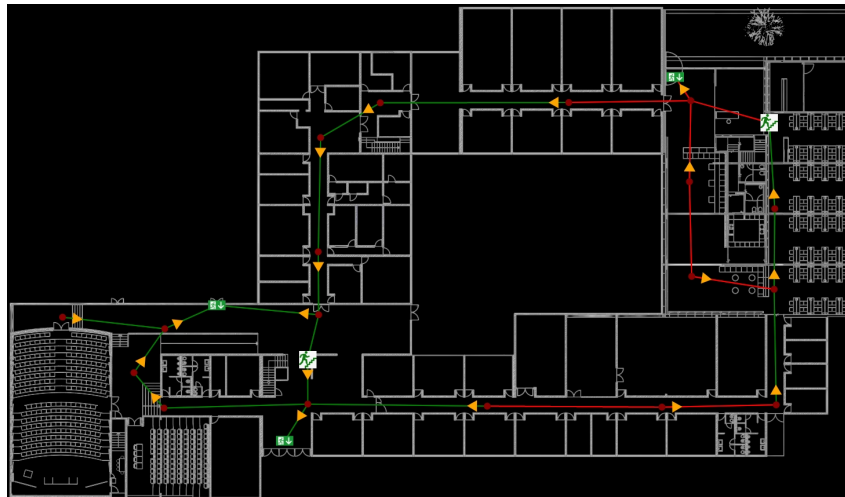


Figure 6.23: Test Case 4: DCC Floor 0 Exit Paths



Figure 6.24: Test Case 4: DCC Floor 1 Exit Paths

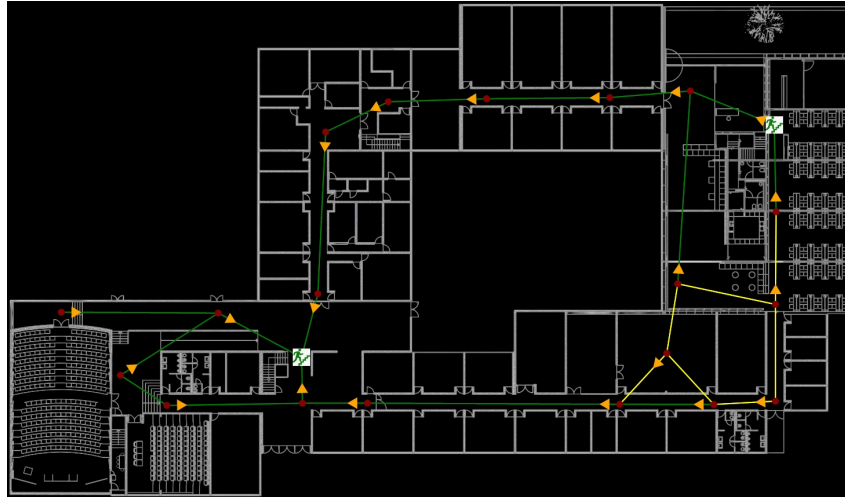


Figure 6.25: Test Case 4: DCC Floor 2 Exit Paths

Using the guidance component, as in the previous experiment, we can confirm in figure 6.26 that the next node to follow by users after node "S27" is node "S51" in floor one. The next node to follow from "S51" is node "S97" from the second floor, which confirms that users near the zone of the node "S27" in floor zero will have to reach the second floor and then return to floor zero in order to exit the building safely.

```

Message received from DCC_Floor0: In the Guidance Node S66 users proceed to Guidance Node S22.
Message received from DCC_Floor0: In the Guidance Node S27 users proceed to Guidance Node S51_DCC_Floor1.
Message received from DCC_Floor0: In the Guidance Node S22 users proceed to Guidance Node S4.
Message received from DCC_Floor0: In the Guidance Node S4 users proceed to Guidance Node S27.
Message received from DCC_Floor1: In the Guidance Node S50 users proceed to Guidance Node S43.
Message received from DCC_Floor1: In the Guidance Node S49 users proceed to Guidance Node S42.
Message received from DCC_Floor1: In the Guidance Node S43 users proceed to Guidance Node S51.
Message received from DCC_Floor1: In the Guidance Node S42 users proceed to Guidance Node S50.
Message received from DCC_Floor1: In the Guidance Node S55 users proceed to Guidance Node S54.
Message received from DCC_Floor1: In the Guidance Node S54 users proceed to Guidance Node S44.
Message received from DCC_Floor1: In the Guidance Node S53 users proceed to Guidance Node S78.
Message received from DCC_Floor1: In the Guidance Node S44 users proceed to Guidance Node S53.
Message received from DCC_Floor2: In the Guidance Node S91 users proceed to Guidance Node S90.
Message received from DCC_Floor2: In the Guidance Node S79 users proceed to Guidance Node S91.
Message received from DCC_Floor1: In the Guidance Node S51 users proceed to Guidance Node S97_DCC_Floor2.
Message received from DCC_Floor2: In the Guidance Node S80 users proceed to Guidance Node S97.
Message received from DCC_Floor2: In the Guidance Node S91 users proceed to Guidance Node S80.
Message received from DCC_Floor2: In the Guidance Node S79 users proceed to Guidance Node S88.
Message received from DCC_Floor1: In the Guidance Node S51 users proceed to Guidance Node S97_DCC_Floor2.

```

Figure 6.26: Next Node to follow by users in Node S27 and Node S51

Figure 6.26 also exposes an interesting aspect: the fact that the guidance node received twice a message indicating that the next node to follow from the Guidance Node "S51" is the Guidance Node "S97_DCC_Floor2". This occurred because the Decision Node that sent this message did not receive the confirmation the first time, which shows that often. Despite the information being successfully delivered, the confirmation message ended up being lost, leading to a retransmission.

6.5 Results Overview

With the explored test cases, we can confirm that the system has a good performance in different conditions, reacting to the values received from the sensors and always finding the safest path in order to users evacuate the building safely. This experience also shows that a shorter time between message retransmission represents an improvement in the system performance, leading to a faster information exchange between components.

It was also shown that the implemented communication protocols worked correctly, retransmitting quickly all the information that was lost, which has proven to be a valuable tool when network conditions are not ideal and we have several lost or delayed packets.

When analyzing the results in weak network conditions, we obtained messages that took around a minute to be successfully delivered, when using a value of five seconds between retransmissions, which represents an acceptable value that can be further improved with a lower value of time between retransmissions. However, this value must take into account the time it takes for a message to be delivered and confirmed, and cannot be so low as to not allow time for these operations to occur. Another aspect is that by increasing the frequency of messages we are increasing network congestion, having the possibility of reaching a point where connections become saturated. The time value between retransmissions is easily configurable in the configuration file and can vary depending on the building topology and typical network conditions.

In conclusion, the obtained results were satisfactory, showing that the system performs well with different network conditions and that it makes the correct decisions regarding the safest path to exit the building.

Chapter 7

Conclusion

In this work, a dynamic system for building evacuation was developed, importing the building information in a standard indoor mapping format, and receiving simulated values from sensors that monitor temperature and smoke/gas values, providing a real-time understanding of what is going on all over the building.

The system is based on a distributed architecture. The building is divided into several areas, where each area is composed by different components that communicate between them and between areas. This means that even when some components fail, the system is still able to operate and also reduces the amount of information that each computational unit has to process, obtaining a better performance which is a huge advantage in real-time applications.

An algorithm to calculate the safest evacuation paths, based on Dijkstra's algorithm, was developed with the major difference consisting in the use of an array of danger level values, instead of a single value, as the cost metric. In this solution, each position represents the distance that has to be walked in a certain level of danger. The selected evacuation path is the one that minimizes the distance traversed in higher danger levels, ensuring that the user is exposed to danger for the minimum possible distance.

The first phase consisted of exploring existing solutions for different parts of the overall system, providing the initial ideas for the development of the system. Choosing one of the available standard formats for indoor mapping was one of the most important decisions, since it allowed us to have all the necessary information about the building to create a computational representation based on which the system can perform different algorithms. The IndoorGML format was chosen, since this standard allows the description of internal areas of the building intuitively, enables the representation of the available paths to users and the identification of different areas in the building. This standard also enabled the representation of the Sensor Network, with the exact location of each sensor, and the assignment of navigation nodes that are covered by each of them.

A unit to process the information from IndoorGML file and make this data available to the other system components was developed. A Server and an API, with the help of a MySQL

database, provide the ability for storing and retrieving this information to the different components in the first phase. Afterward, they will store this information locally to always have it available, even when no communication to the server is possible.

Though the system uses different types of nodes, this work was more focused on the Decision Nodes, where the information and decisions of each area are processed, running different algorithms to know what information needs to be sent to Decision Nodes in neighboring areas and to find the safest path in its area. Communication protocols were developed to address the messaging requirements among the different components and the problems of retransmissions between them in sub-optimal network conditions.

Regarding Sensor Nodes and Guidance Nodes, for the former an application was developed that simulates the values read by each sensor in the building, allowing to apply and test the communication protocol and understand how the Decision Node reacts when receiving information from the sensors. In the case of Guidance Nodes, an application was developed that simulates all Guidance Nodes in the building, where each node receives the next node in the path to be followed by evacuees. This solution allowed us to quickly and easily apply and test the efficiency of the communication protocol between Decision Nodes and Guidance Nodes.

In conclusion, analyzing the proposed objectives for this thesis, it was developed a system capable of receiving a standard indoor mapping format, IndoorGML being the chosen format. The system uses sensors to monitor the building conditions, can use the information provided by them to understand which zones are affected and is capable of calculating the safest path to evacuate the building. A hybrid architecture allows the system to work even when a component fails, with the building divided into different areas and the information distributed by Decision Nodes present in these areas. A mechanism that allows reliable information exchange between Decision Nodes was developed, allowing them to have a global view of the building which is essential to compute the safest evacuation path. So that the components do not depend on a connection to the central server, it was developed mechanisms of local storage, allowing the system to work even when there is no connection to the server. Different interfaces were also studied to indicate to users the direction to follow, Guidance Nodes, exploring the pros and cons of each one and how they can be combined to transmit this information efficiently and reduce evacuation time. In summary, all the objectives proposed for this work were successfully achieved.

7.1 Future Work

Although the project reached very satisfactory results, it is worth noting that there are potential enhancements that were not incorporated into the project. These omissions are primarily attributed to time constraints during the implementation phase. Therefore, this subsection will outline some of the prospective improvements that could be considered in future work.

7.1.1 Congestion of evacuation paths

An important but complex feature would be adding some type of load balancing to the evacuation path, avoiding that, in buildings with many users, they are sent in large numbers along the same route, which could compromise the efficiency of evacuation. A mechanism that could balance and guide users through different paths, as long as it does not increase the level of danger they are exposed to, would be beneficial for a system of this nature. A possible solution would be something similar to what was used in [16], where a camera was installed in the sensor node, allowing to understand the density of users in a certain area and affecting the weight of the edge in the navigation graph. This solution would require a careful approach, since the use of dynamic metrics in routing is known to lead to instability.

7.1.2 How to recover from system reboots

During an emergency, reboots in the components can occur. In the future, it would be nice to explore and implement a mechanism allowing them to quickly restore the previous state, not losing important information across reboots.

7.1.3 Security Mechanisms

Another enhancement, that must be implemented before such a system goes into production, involves the security of the application. It is necessary the introduction of authentication tokens to secure access to the API. Currently, there are advanced developments in this subject that could not be completed within the project's time constraints. So, right now, the API is missing the necessary protection measures. The communication between components should also be protected due to, for example, the effects that an attack where message values are changed can have on the system.

7.1.4 Guidance Nodes Interface

This work addressed the part of the communication between Decision Nodes and Guidance Nodes and presented some solutions for the interface that communicates to users the direction to follow, exploring the advantages and disadvantages of each one. Due to the importance of this component, finding the right solutions requires a lot of research, but in the future the implementation of interfaces would be valuable.

7.1.5 Duplicated Decision Node

In the proposed solution, only one Decision Node is considered by area. In the future, it would be great to have at least one second Decision Node which is a copy of the original one. This

component would represent a precautionary measure for when the first one stops working, which could happen due to the conditions that exist during hazards. Redundancy would ensure that the system continued to work correctly and no information was lost even in the presence of hardware failure, increasing the robustness and, consequently, the effectiveness of the system.

Bibliography

- [1] Adafruit. [Dht11, dht22 and am2302 sensors overview](#), Accessed May 2023.
- [2] Apple. [Indoor mapping data format overview](#), Accessed May 2023.
- [3] Matthew Barnes, Hugh Leather, and D.K. Arvind. [Emergency evacuation using wireless sensor networks](#). In *32nd IEEE Conference on Local Computer Networks (LCN 2007)*, pages 851–857, 2007. doi:10.1109/LCN.2007.48.
- [4] H. Butler, M. Daly, A. Doyle, S. Gillies, D. Herring, M. Hooten, R. Lloyd, J. Schaad, and M. Shepherd. [The gejson format \(rfc 7946\)](#). Technical report, IETF, 2016.
- [5] MappedIn Developers. [Mappedin](#), Accessed May 2023.
- [6] Abdoulaye A. Diakit , Sisi Zlatanova, and Ki-Joune Li. [About the subdivision of indoor spaces in indoorgml](#). *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, pages 41–48, 2017.
- [7] EATON. [Fire safety in high risk buildings – preventing the next tragedy](#), 2019.
- [8] Last Minute Engineers. [How mq2 gas/smoke sensor works? interface it with arduino](#), Accessed May 2023.
- [9] Esri. [Arcgis indoors](#), Accessed May 2023.
- [10] Esri. [An introduction to indoor mapping with arcgis](#), Accessed May 2023.
- [11] Avgoustinos Filippoupolitis and Erol Gelenbe. [A distributed decision support system for building evacuation](#). In *2009 2nd Conference on Human System Interactions*, pages 323–330, 2009. doi:10.1109/HSI.2009.5091000.
- [12] GeeksForGeeks. [Types of nosql databases](#), Accessed May 2023.
- [13] Google. [What is a relational database?](#), Accessed May 2023.
- [14] Hugo Ledoux. [val3dity tool](#), Accessed May 2023.
- [15] Van-Quyet Nguyen, Huu-Duy Nguyen, Quyet-Thang Huynh, Nalini Venkatasubramanian, and Kyungbaek Kim. [A scalable approach for dynamic evacuation routing in large smart](#)

- buildings. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 292–300, 2019. doi:10.1109/SMARTCOMP.2019.00065.
- [16] Van-Quyet Nguyen, Huy-The Vu, Van-Hau Nguyen, and Kyungbaek Kim. [A smart evacuation guidance system for large buildings](#). *Electronics*, 11(18), 2022. ISSN: 2079-9292. doi:10.3390/electronics11182938.
- [17] ns-3 Development Team. ns-3: A discrete-event network simulator for internet systems. <https://www.nsnam.org/>, 2021.
- [18] nsnam. [Ns3 topology generator](#), Accessed May 2023.
- [19] OGC. [Indoorgml](#), Accessed May 2023.
- [20] Faisal Saeed, Anand Paul, Abdul Rehman, Won Hwa Hong, and Hyuncheol Seo. [Iot-based intelligent modeling of smart home environment for fire prevention and safety](#). *Journal of Sensor and Actuator Networks*, 7(1), 2018. ISSN: 2224-2708. doi:10.3390/jsan7010011.
- [21] Seedstudio. [Grove - gas sensor\(mq2\)](#), Accessed May 2023.
- [22] STEMLab. [Ineditor](#), Accessed May 2023.
- [23] Soni Upadhyay. [Bellman-ford algorithm: Pseudocode, time complexity and examples](#). Last accessed in May 2023.
- [24] Y. Zhang, D. Zhou, S. Chen, S. Gao, and Y. Ma. Single-image crowd counting via multi-column convolutional neural network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 589–597, Las Vegas, NV, USA, June 2016.