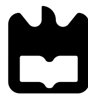
Universidade de Aveiro
2015

Departamento de Eletrónica,
Telecomunicações e Informática

RUI
REBELO
BRITO

INTELLIGENT DATA TRANSFER FOR DIGITAL ASSEMBLY INSTRUCTIONS

TRANSFERÊNCIA INTELIGENTE DE INSTRUÇÕES DIGITAIS DE MONTAGEM


Universidade de Aveiro
2015

Departamento de Eletrónica,
Telecomunicações e Informática

RUI
REBELO
BRITO

INTELLIGENT DATA TRANSFER FOR DIGITAL ASSEMBLY INSTRUCTIONS

TRANSFERÊNCIA INTELIGENTE DE INSTRUÇÕES DIGITAIS DE MONTAGEM

Dissertação apresentada à Universidade Técnica de Hamburgo para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Sistemas de Informação na Universidade de Aveiro, realizada sob a orientação científica do Doutor Hermann Lödding, professor do Instituto de Gestão de Produção e Tecnologia da Universidade Técnica de Hamburgo, e do Doutor Cláudio Teixeira, professor auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

the jury

First Examiner	Prof. Dr. Hermann Lödding Hamburg University of Technology
Second Examiner	Prof. Dr. Cláudio Teixeira Universidade de Aveiro

acknowledgments

I would like to express my gratitude to Prof. Herman Lödding, without whom I would have never been able to take advantage of this opportunity. I am extremely grateful to Prof. Cláudio Teixeira, who always made himself available to support this work. All the conversations and advice were extremely helpful for the good course of this thesis. I would also like to show my gratitude to my coordinator Phillip, whose guidance was instrumental to overcome most of the challenges I faced during this work.

A special thank you to Alex, for his help and availability in brainstorming ideas.

I would like to thank my special friends Cris, André, Andreas, Renato, and Eduardo, whom despite the distance were always close, their support meant the world to me.

To Semah, Lino, and Sandro, I can not begin to quantify how their presence got me going when facing many adversities.

I would like to deeply thank my mother, father, and sister, who always did everything they could and could not do to support me. I aspire to be the kind of person they are.

Last but not least, words cannot express how deeply thankful I am to Rita for her continuous support and help. It will be something I will never forget.

keywords shipbuilding, data transmission, auguemented reality.

abstract

The German maritime industry has a strong economic influence, prevailing over competitors by offering a higher quality product and a higher degree of flexibility during design and building phases. In order to maintain its competitive edge, means to increase the productivity of a ship's construction process are researched and developed. Under the PROSPER project, an Android application running augmented reality technology was developed to assist shipbuilders in the performance of their tasks. In this work, we researched organizational models capable of holding assembly instruction information later transfered to the application. Different wireless technologies and protocols were analyzed and studied to determine the most suitable transferring mechanisms.
In the end, a Node.js server capable of handling complex assembly instructions requests was developed and tested using real-word scenarios. Throughout the development process, time was devoted to security measures in the protection of the server, end connections, and all information. Two different transferring methods were developed and testes, together with improvements to the overall transfer of assembly instructions.

palavras chave

construção naval, transmissão de dados, realidade aumentada.

resumo

A indústria marítima alemã tem uma forte influência econômica, prevalecendo sobre os seus concorrentes, oferecendo um produto de maior qualidade e um maior grau de flexibilidade durante as fases de conceção e construção. A fim de manter a sua vantagem competitiva, fins para aumentar a produtividade do processo de construção de um navio são investigados e desenvolvidos. No âmbito do projecto PROSPER, uma aplicação Android foi desenvolvida para ajudar, através de realidade aumentada, construtores navais no desempenho das suas funções. Neste trabalho, investigamos modelos organizacionais capazes de armazenar informação pertinentes a instruções de montagem de navios, mais tarde transferidas para a aplicação. Diferentes tecnologias e protocolos sem fio foram analisados e estudados para determinar os mecanismos mais adequados para o procedimento de transferência. No final, um servidor Node.js capaz de lidar com complexas solicitações de instruções de montagem foi desenvolvido e testado utilizando cenários reais de montagem. Durante todo o processo de desenvolvimento, medidas de segurança na proteção do servidor, conexões entre servidor e clientes, e informações foram implementadas. Dois métodos distintos de transferência foram desenvolvidos e testados, juntamente com melhorias para a transferência global de instruções de montagem.

CONTENTS

1 INTRODUCTION 1
- 1.1 Context 1
- 1.2 Thesis Objectives 2
- 1.3 Thesis Structure 4

2 PROBLEM STATEMENT AND CONCEPT 5
- 2.1 State of the Art of Ship Building 5
- 2.2 Problem Description 12
- 2.3 Proposed Solution 12
- 2.4 Decision Support System 15

3 ARCHITECTURE 17
- 3.1 Solution Constraints 17
- 3.2 Data Transmission 20
  - 3.2.1 Transport Protocol 21
  - 3.2.2 Transferring Protocol 24
- 3.3 Security 27
  - 3.3.1 Access Control 27
    - 3.3.1.1 Authentication 27
    - 3.3.1.2 Authorization 28
  - 3.3.2 Secure Communication 29
  - 3.3.3 Data Protection 30
  - 3.3.4 Access and Request Authentication 32
- 3.4 Relational Database 33
  - 3.4.1 Model Representation 34
  - 3.4.2 Password Protection 34
  - 3.4.3 Assembly Instructions Storage 36
- 3.5 System Abstraction 37
- 3.6 Assembly Instruction Transfer 39
  - 3.6.1 Work Package List 40
  - 3.6.2 Data Selection 41
  - 3.6.3 Transfer Methods 48
    - 3.6.3.1 Condense Method 48
    - 3.6.3.2 Multiple-Requests Method 49
    - 3.6.3.3 Method Comparison 50
  - 3.6.4 Transfer Time Improvement 51
    - 3.6.4.1 Location Identifier 51

3.6.4.2 File Caching . . . 52
3.6.5 Instructions Update . . . 53
3.7 Service Mapping . . . 54

4 FRAMEWORK 57
4.1 Database and Datastore . . . 57
4.1.1 MySQL . . . 57
4.1.2 Redis . . . 57
4.2 Server Framework . . . 58
4.2.1 Node.js . . . 58
4.2.2 Node Package Manager . . . 59
4.3 Client Framework . . . 60
4.3.1 Android Operating System . . . 60
4.3.2 Android HTTP Client . . . 61

5 IMPLEMENTATION 63
5.1 Database Implementation . . . 63
5.1.1 Database Structure . . . 63
5.1.2 Work Package Dependencies . . . 65
5.1.3 Data Validation . . . 67
5.2 System Configuration . . . 69
5.2.1 System Modular Design . . . 69
5.2.2 Server Configuration . . . 70
5.2.2.1 Package Declaration . . . 70
5.2.2.2 Redis and MySQL Connection . . . 71
5.2.2.3 Sequelize Models . . . 73
5.2.2.4 Application Configuration . . . 74
5.2.2.5 Request Routes and Authentication . . . 75
5.2.2.6 Messages Format . . . 76
5.2.2.7 Redis Database Structures . . . 77
5.2.3 Client Configuration . . . 79
5.2.3.1 Package Declaration . . . 79
5.2.3.2 Android HTTP Client Interface . . . 80
5.3 Assembly Instructions Updates Method . . . 81
5.4 Transfer Methods Implementation . . . 83
5.4.1 File Encryption . . . 83
5.4.2 Condense Method Implementation . . . 83
5.4.3 Multiple-Requests Method Implementation . . . 85
5.5 Test Results . . . 89
5.5.1 Transfer Methods Comparison . . . 89
5.5.2 Load Testing . . . 90

6 CONCLUSION 93

BIBLIOGRAPHY 95

COMMUNICATION CHANNELS 105

LIST OF FIGURES

Figure 1.1	Prototype display	2
Figure 1.2	Manual transmission of information	3
Figure 1.3	Automatic Wireless transmission of information	3
Figure 2.1	Ship production/construction stages	5
Figure 2.2	Initial ship sketchs	6
Figure 2.3	Ship assembly process	7
Figure 2.4	Shipyard layout	7
Figure 2.5	Sub assembly construction process of a ship's unit	10
Figure 2.6	Model complete Unit	11
Figure 2.7	Model unit definition	11
Figure 2.8	Model block assembly analysis	11
Figure 2.9	3D solid CAD model	11
Figure 2.10	Work Package	13
Figure 2.11	Work package dependencies structure	14
Figure 2.12	Model proposal	15
Figure 3.1	TCP sequence diagram	23
Figure 3.2	SSL connection establishment	31
Figure 3.3	Model's entity-relationship diagram	35
Figure 3.4	Authentication Process	37
Figure 3.5	System abstract overview	38
Figure 3.6	Block and Section Assembly Order	42
Figure 3.7	Ship Example	43
Figure 3.8	Work Package ship structure	43
Figure 3.9	Condense Method Process	48
Figure 3.10	Distinct TCP connections per request	49
Figure 3.11	Mutil Request Connection	50
Figure 3.12	Cache Writing Policy	52
Figure 3.13	Periodic Instructions Update	53
Figure 4.1	Android Activities Life Cycle	61
Figure 5.1	MySQL Database	64
Figure 5.2	Trigger Association Validation	68
Figure 5.3	System abstract overview	69
Figure 5.4	Redis data structures	78
Figure 5.5	HTTP packets capture.	88
Figure 5.6	Methods comparison	90
Figure 5.7	Load test total time	91

Figure .1 Simplex channel 105
Figure .2 Half-Duplex channel 105
Figure .3 Full-Duplex channel 105

LIST OF TABLES

Table 3.1 802.11 Specifications Comparison 19
Table 3.2 TCP and UDP comparison 22
Table 3.3 TCP acknowledge and sequence packets 23
Table 3.4 Password Hashes 36
Table 3.5 Password Hashes and Salt 36
Table 3.6 Service Mapping 55
Table 5.1 Hierarchical data designs comparison 66
Table 5.2 Closure Pattern of S" 66
Table 5.3 Method results values 89
Table 5.4 Load test summary results 91

LISTINGS

Listing 1 List Work Package Example 41
Listing 2 List of Parts for U5 dependecy selection 46
Listing 3 Records Association Trigger 68
Listing 4 Package.json 71
Listing 5 Redis Client 72
Listing 6 Sequelize Client 72
Listing 7 Sequelize Blueprint Model 73
Listing 8 Sequelize FindbyId BluePrint 74
Listing 9 Express App configuration 74
Listing 10 Node.js Server 74
Listing 11 Route GET request 75
Listing 12 Mysql.json 77
Listing 13 Login method 78
Listing 14 build.gradle 79
Listing 15 Client HTTP Interface 80
Listing 16 Update Service 81
Listing 17 AlarmManager 82
Listing 18 Alarm scheduler 82
Listing 19 Encrypt Function 83
Listing 20 Condense Method Service 84
Listing 21 Android condense method 84
Listing 22 Android Multi-Request method 85
Listing 23 Multi-Request Method Service 86

ACRONYMS

2D	Two-Dimensional.
3D	Three-Dimensional.
ACL	Access Control List.
API	Application Program Interface.
AR	Augmented Reality.
BLE	Bluetooth Low Energy.
BLOB	Binary Large OBject.
CA	Certificate Authority.
CAD	Computer-Aided Design.
CTO	Chief Technology Officer.
DRM	Digital Rights Management.
ER	Entity-Relationship.
HMAC	Hash Message Authentication Code.
HTTP	Hypertext Transfer Protocol.
I/O	Input/Output.
IEEE	Institute of Electrical and Electronics Engineers.
IP	Internet Protocol.
IR	Infrared.
JSON	JavaScript Object Notation.
MAC	Message Authentication Code.
MVC	Model-View-Controller.
NIST	National Institute of Standards and Technology.

npm Node Package Manager.

OHMD Optical Head-Mounted Display.
ORM Object-Relational Mapper.
OS Operating System.

PIN Personal Identification Number.

RDMS Relational Database Management System.

SDK Software Development Kit.
SHA Secure Hash Algorithm.
SQL Structured Query Language.
SSE Server-Sent Events.
SSL Secure Sockets Layer.

TCP Transmission Control Protocol.
TLS Transport Layer Security.
TUHH Hamburg University of Technology.

UDP User Datagram Protocol.
URI Uniform Resource Identifier.
URL Uniform Resource Locator.
UTC Coordinated Universal Time.

WLAN Wireless Local Area Network.
WPA Wi-Fi Protected Access.
WPA 2 Wi-Fi Protected Access II.
WPAN Wireless Personal Area Network.
WWAN Wireless Wide Area Networks.
WWW World Wide Web.

XML Extensible Markup Language.

1 INTRODUCTION

The German maritime industry has a strong economic influence, prevailing over competitors by offering a higher quality product and a higher degree of flexibility during design and building phases. They specialize in the production of passenger ships, yachts, ferries, and naval vessels, each with its own unique size, shape, and configuration. To maintain this lead, the industry needs to increase the productivity.

From planing, to production, to sea trials, building a ship is a laborious process that involves a substantial amount of time, precision, and several calculations on the ship's shape optimization, noise and vibrations, propellers, ducts, and paddles. Additionally, rules, certificates and other legalities increase the complexity of this task. There are numerous approaches to shipbuilding, such as synchronized assembly lines for ship sections, building sites for larger blocks of the ship, and on-board equipment. Detailed assembly schedules and drawings for every ship are essential for the construction. Examining these drawings, discussing and distributing the work that will be performed requires a great amount of time. [1] To help resolve uncertainties in the design, workers and foremen consult interactive stationary Three-Dimensional (3D) Computer-Aided Design (CAD) terminals together with the traditional Two-Dimensional (2D) drawings. These methods of operating are not enough to maintain a competitive edge, since there is still a lot of time spent in gathering the information needed for assembly. This is aggravated by the fact that often, during the construction phase, clients have changes of heart regarding their needs. Alterations to the initial design, resulting from the need to improve it or correct possible flaws, have to be implemented and properly tested, forcing workers to restarting the process of gathering and understanding the new information all over again.

1.1 CONTEXT

In order to increase the productivity during the building phase of ships, one sub goal of the research project PROSPER is to provide digital assembly instructions *via* an application on mobile devices operated by shipbuilders, while they perform their tasks. This application would provide all the necessary information, reducing the gathering process and the time spent understanding the task. Devices like tablets or Optical Head-Mounted Displays (OHMDs) are a cheap and viable solution to achieve this goal. Having a single point where all the assembly instructions, including both traditional 2D and interactive 3D, can be gathered and explained, results in a faster understanding of the necessary assembly steps.

To achieve the goals set by project PROSPER, the Augmented Reality Laboratory at the Hamburg University of Technology (TUHH) has developed an Android prototype, dubbed ***Digital Assembly Assistant***, that makes use of smart technologies like Augmented Reality (AR), where computer elements are augmented (or added to) by input sensors, enhancing the current perception of reality. The idea is to connect shipbuilders and take advantage of a "fusion between already assembled components and to-be installed parts" [1] making it easier to understand how components must be assembled. Figure 1.1 depicts a screenshot of the current application's display, where it is possible to see the available options, as well as the components that have to be assembled. With this application a shipbuilder can have access to the information, and have them displayed in the real environment, as well as the CAD model itself.

For a more detailed examination of the current prototype, we recommend reading the literature published under the PROSPER project, particularly [1].

1.2 THESIS OBJECTIVES

At the current stage, the pertinent information for a specific assembly, still has to be manually gathered and transfered to the device (Figure 1.2). This can be inefficient and time consuming, particularly when design changes occur frequently. An essential requirement, in any construction, is to have the most up-to-date information so that the component can

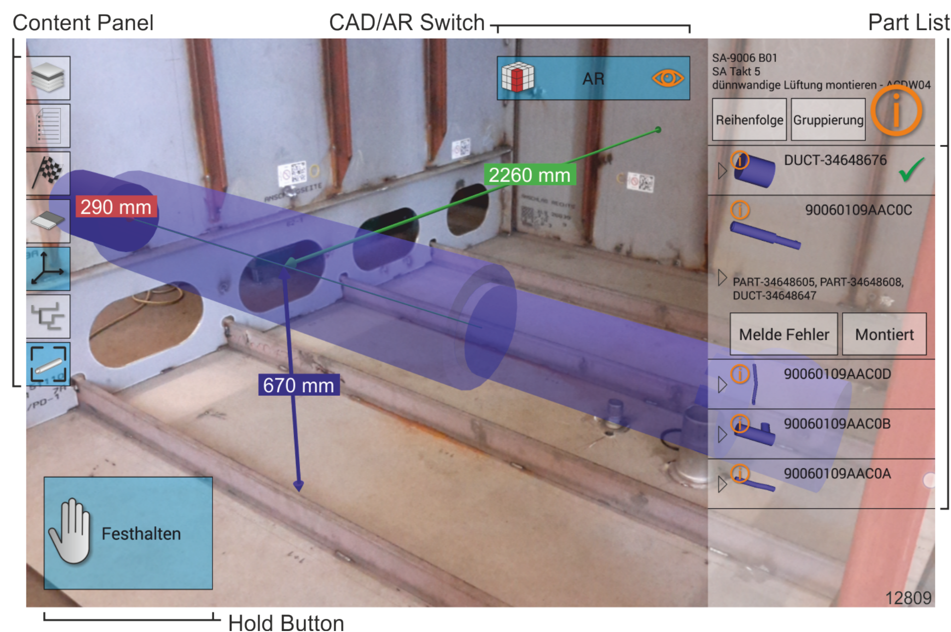


Figure 1.1: Screenshot of the current prototype. The display demonstrate a real-live scenario showing a mounting pipe scenario, with components highlighted in blue (AR).

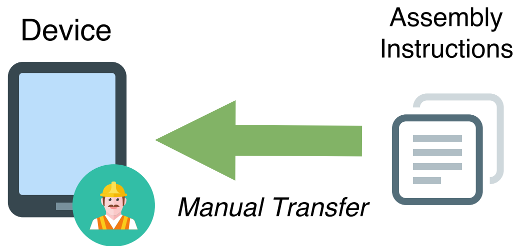


Figure 1.2: Prototype current state: Manual selection and transfer of information.

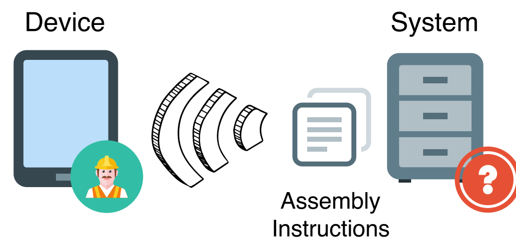


Figure 1.3: Automatic Wireless transmission of information.

be assembled. Thus, a content provider system (Figure 1.3) that can seamlessly and automatically transfer the necessary information to the application must be established/put in place. Also, in order to take full advantage of the devices' portability, a precondition is having the system and the devices communicate wirelessly.

There are some challenges in the distribution of digital assembly instructions to these devices. Without a comprehensive understanding of the assembly process, one can not foresee the relevant information for a worker, at a particular assembly step. Determining how, when, and which data needs to be transfered, and how the correlation between a specific mobile device and the to-be transferred data is made, represents a serious challenge that can undermine what this research project is trying to accomplish. Moreover, we must only take in consideration technologies generally available in most of today's mobile devices, and consider the usability, performance, and battery life at every planning and implementation stage.

This thesis has two primary objectives: First, to find a model in which assembly instructions can be organized by several forms/aspects/ and correlated by specific task, worker, and type of construction. Second, to design a system that transmits assembly instructions data to mobile devices used by (groups of) workers for information gathering. For this purpose, it has to be determined how, when, and which data has to be transfered, and how the correlation between a specific mobile device and the to-be transferred data is made.

To achieve these objectives a precise analysis of the situation and analytic specification of the problem must be conducted. This step is necessary to fully understand the current state of the problem in study, and subsequently propose a suitable solution for it.

In order to develop the system for holding and transmitting the information to the devices, an evaluation/discussion of the current data transmission protocols and techniques, as well as a study of the current assembly scenarios in shipbuilding, must be conducted. In addition, considerations must be made for technologies and features present in most mobile devices. In order to develop the system's concept, we need to define, based on the correlation between worker, device, component, work package, and data, the necessary data to be transfered. We also need to detect design changes on the data currently in use and implement a back channel to confirm that data. Finally, we must validate the concept

by implementing a software prototype. For the test purposes, Mayer Shipyard supplied a assembly instructions and task plans of a ship's section.

1.3 THESIS STRUCTURE

This thesis is organized as follow: Chapter 2 describes the state of current maritime building and manufacture, intending for the reader to get familiarized with the industry revolving the topic. The development of the industry and the role of modern tools, like CAD models, are briefly explaining in this chapter. Moreover, the problem present in how information is gathered and distributed to workers. A organization model capable of holding pertinent information to the assembly instructions is also introduced in this chapter.

Chapter 3 describes the architectural solutions and design plans that, based on the objectives specified in Section 1.2 and the model defined in Chapter 2, were drafted in order to achieve the presented solution. A large amount of time and research was devoted in determining the best wireless technologies and protocols, transmission mechanisms, and security measures to be used in the implementation of our solution. In Chapter 4, we introduced the different framework and libraries used to implement our backend server.

In Chapter 5 we demonstrate how using these frameworks, the presented solution was implemented, following the plans detailed in Chapter 3. Still in this chapter, we present the test results of the transferring mechanisms and the overall system.

Finally, the conclusions of this work are outlined in Chapter 6, as well as possible future work.

2 PROBLEM STATEMENT AND CONCEPT

In this chapter we uncover the problem of developing a model that can uphold the information structure necessary to run the current application. In order to fully understand the concept, the ship assembly process is analyzed together with the proposed model and the constraints that restrict it.

2.1 STATE OF THE ART OF SHIP BUILDING

One of the main objectives of this thesis is to design and create a model that can be used to organize assembly instructions, later distributed to workers in an easy and efficient way.

Since this model would ideally be adopted by all shipyards, and other industries perhaps, it has to be easily accessible and general enough, to be adapted in different contexts, but still cover all the fundamental requirements.

In order to reach this model, we must first analyze the present situation in (today's) shipyards, in particular how ships are designed and planned, and how the assembly instructions are assigned to workers. The first step is to understand the stages in the production/construction of a ship. A comprehensive view of these stages is depicted in Figure 2.1.

The product teams: *Engineering, Planning, Purchasing* operate in stages *prior* to the actual construction of the ship. It is during those stages that negotiations between the shipyard and the shipping company take place; initial sketches, like the one showed in Figure 2.2, are drawn throughout the negotiation process. These preliminary sketches are drawn based on the shipping company requirements, and illustrate the idea for the final product. Moreover,

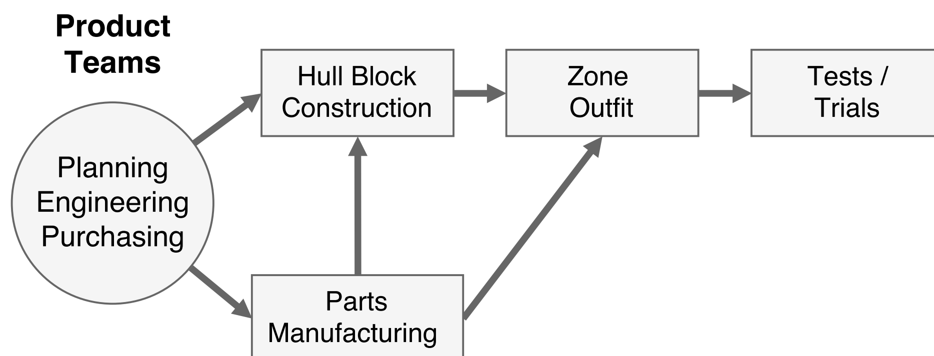


Figure 2.1: Ship production/construction stages.

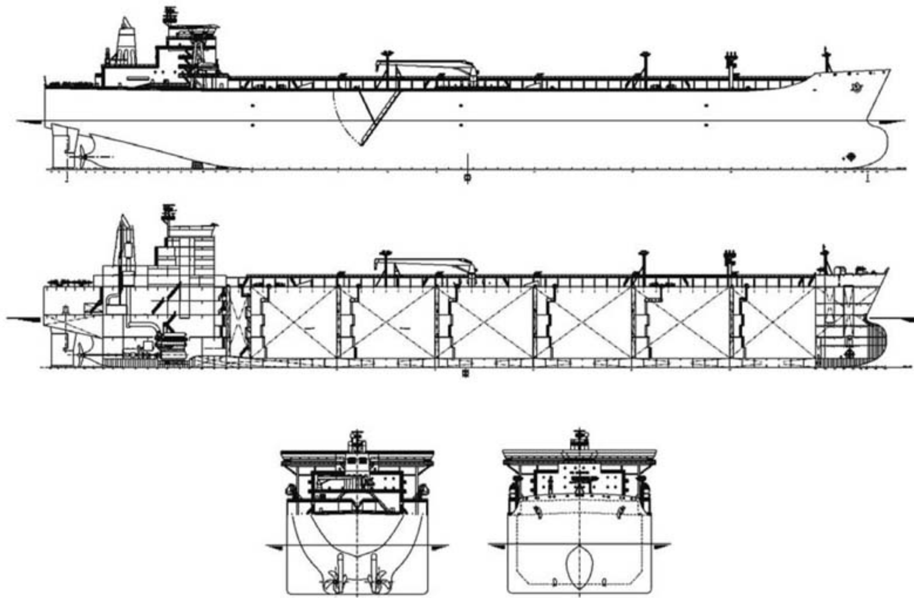


Figure 2.2: Initial ship sketchs (Taken from: [3]).

they can also show the dimensions of the ship, total capacity of the hold, delivery time previsions, as well as the required materials and the estimated budget.

"More and more ships are built in larger series on a standard design, with limited variations between the different ships" [2]. Shipyards have a diverse range of standardized ships, with shipping companies ordering whole series with only few modifications at an extra cost. [4]. The use of a standard design allows shipyards to make ships in a series-production, decreasing costs and increasing production. Standard designs also give the shipping company a better idea of the final result and cost; the design should have already proven reliable and any flaws in the design should have been rectified.

In the end, the overall construction time decreases, as both the planning and the building phase are considerably shorter - plans from previous projects can be used as a starting point and workers are already familiarized with the basis of the design. By saving time, shipyards save money and can offer the same product at a lower cost for/to the clients.

After the negotiations are over, the *Planning* and *Engineering* teams start preparing the final design for the ship, following the specifications set in the contract. This task requires extensive calculations over a long period of time, especially if the design is entirely new. A considerable number of man-hours goes into this stage, where the design is planned in order to obey mandatory regulations of security and several other engineering requirements. The final design has to include elaborated and detailed schematics of the construction plans for the ship's systems, such as the mechanical, hydraulic, pneumatic, and electrical systems. [5] The *Production* and *Purchasing* teams receive the construction drawings from the *Engineering* team, and can start preparing the materials ordering list and the job-descriptions for

the ship's construction. The planning and managing of the ship construction requires a careful coordination of a wide range of different resources and responsibilities.

Before welding came into wide-scale use in the 1930s, every ship was constructed on the building berth [1], the keel [2] laid and the ship was built upwards, in the same place. [8]
The modern method is to assemble large parts of the ship, each one of them built from smaller sub-assembled parts or components. The large parts of the ship are then brought together to form the complete bow or stern. Finally, these parts are welded together to form the complete ship. A typical assembly process can be seen in Figure 2.3. Sections of the ship are generally manufactured in large buildings, before being transported to the building berth. Once there and following the schematics, they are fitted into place and welded to the adjacent component, section, or block. Figure 2.4 depicts a fictional shipyard layout where the ship assembly flow is illustrated by black arrows. Components, parts, sections, and blocks are built separately and then transported to the building berth, using cranes or other vehicles. However, a piece or component do not necessarily need to go through all the assembly stages, and can be mounted directly onto the ship, as shown in Figure 2.3.

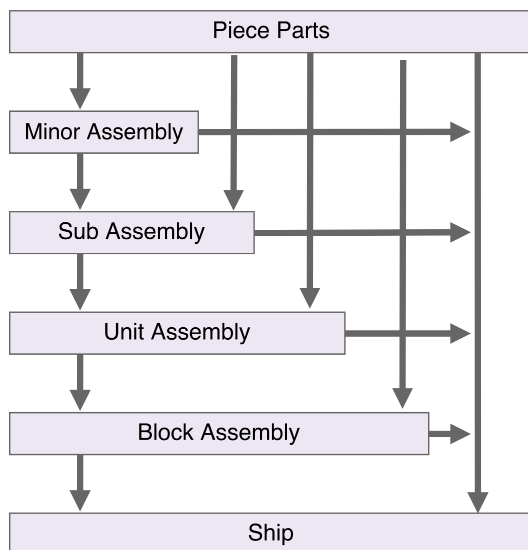


Figure 2.3: Ship assembly process.

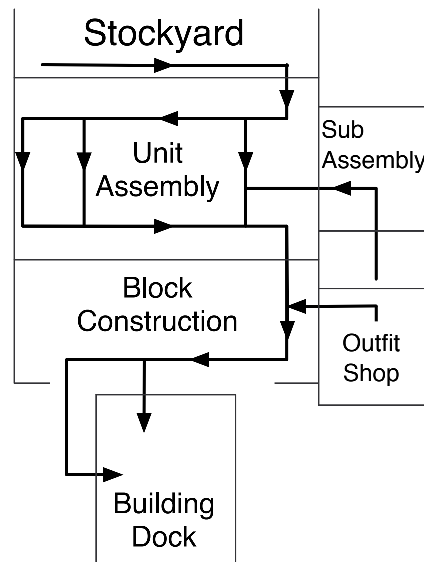


Figure 2.4: An example of a fictional shipyard layout.

This method of construction is usually referred as **module block construction**. The advantages of this procedure are that "work can proceed under cover, unhampered by bad

1 Berth — a location where a ship can be anchored or moored. [6]
2 Kell — a structural beam in the bottom of a hull, extending from the bow to the stern, serving as the foundation of the ship. [7]

weather, and the units or component parts can be built up in sequences to suit the welding operations — not always possible at the building berth itself" [8]; Greater flexibility and reuse as modules can be disassembled and relocated or refurbished; There is less site disturbance as traffic is greatly minimized; Elimination of Weather Delays as great part of the construction takes place inside a building; The indoor construction environment reduces the risks of accidents and related liabilities for workers; And since construction can occur simultaneously in different sites — in parallel — there is an increase in construction's speed.

The advantages of the **module block construction** technique are tremendous, unpredictable circumstances like the weather no longer have such a significant impact in the construction progress. Under the module block construction technique, assembly operations can be roughly divided into the following three points:

- **Assembly on Outfit Unit:** is the most productive (in terms of time) stage of the construction, where smaller units are generally built that can be later outfitted into the hull.
- **Assembly on Hull Block:** less productive than the previous, but more so than the assembly on board ship. At this point, previously assembled units are outfitted into the hull.
- **Assembly on Board Ship:** it is the least productive stage of the construction, as it requires previously built units or blocks, which can suffer from delays due to necessary incomplete units.

To recap, with any complex structure, such as the one of a ship, the assembly is better achieved by splitting work into smaller interim tasks. [9] In Figure 2.5 we can see all the components and minor assemblies that form part of a ship's unit. By dividing the construction of the components into smaller tasks, construction can take place in parallel and in different locations. In the end, the complete unit is transported to the next stage/location, where it can be fitted to a block, asserting the advantage of a modular construction. In Figure 2.6 we can see the unit and four others that when fitted together form the ship's unit. This unit and others form a ship's block, Figures 2.7, 2.8, and 2.9. This block is them transported to the building berth where it is welded to the keel, hull, or other blocks of the ship.

Figure 2.5: Sub assembly construction process of a ship's unit, separated by minor assemblies and their parts/components. (Taken from [9])

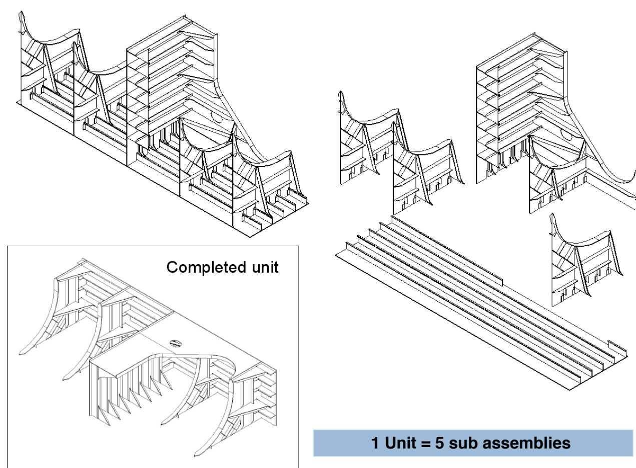


Figure 2.6: Model of a complete unit and its five assembly parts. (Taken from [9])

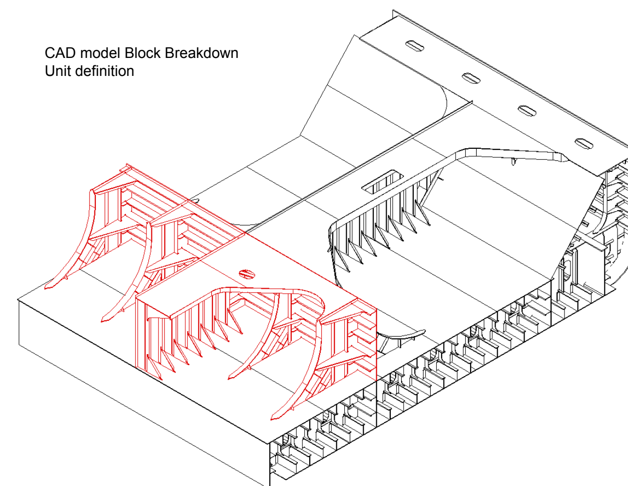


Figure 2.7: Model unit definition. (Taken from [9])

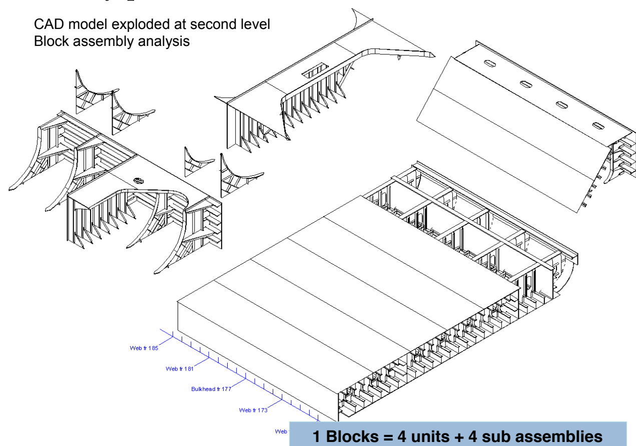


Figure 2.8: A block assembly analysis, featuring four units and four assemblies. (Taken from [9])

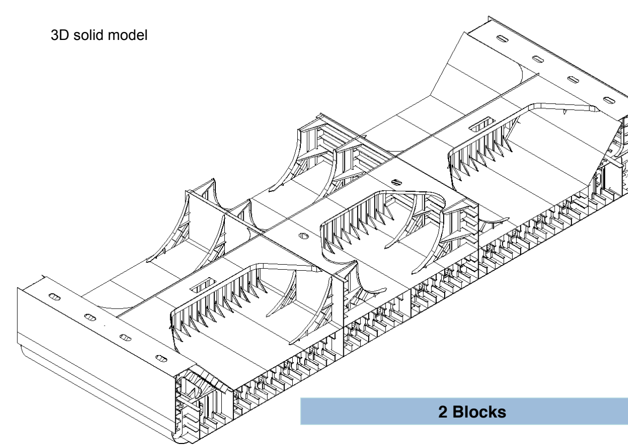


Figure 2.9: 3D solid CAD model of the hull of a ship. (Taken from [9])

2.2 PROBLEM DESCRIPTION

Up to this point, we have been discussing a ship's assembly process. Figures 2.5 to 2.9 should provide a clear visual understanding of how ships are built under the module block technique typically used today.

Assembly operations are the most cost driven and are largely influenced by whom and when the assembly is performed, planning must be done carefully. In the previous section, we explained how careful assembly planning can have a big impact in the ship's construction productivity. Different procedures and techniques exist that strive to improve this process, by reducing the overall complexity and accelerating the construction. However, these improvements can be impaired by poor management skills when distributing assignments among workers.

In modern shipyards, the planning team has access to a variety of tools and optimization algorithms that have been developed and contextualized, through extensive research and studies, to determine the best space allocation and scheduling for the construction. [10] [11] [12] [13] The problem lies in the medium in which these task plans are distributed, which are currently in the form of massive spreadsheets. These spreadsheets cover the pertinent information related to a certain task, such as the assembly order, the materials/parts needed. However, this information is for the most part in-house references or identification codes, which can proven to be complicated and confusing for to understand due to the abundance of information, most of which not being necessary when distributing tasks. Next, is the responsibility of the project managers or foremen, who supervise the construction, to manage the workers and assign individual tasks. Afterwards, all necessary diagrams/drawings must be gathered and understood before the actual construction can start/take place.

2.3 PROPOSED SOLUTION

As mentioned in Chapter 1, the primary objective of the PROSPER project is to reduce/-optimize the time spent on such preparatory activities. In this thesis, we are addressing this problem by having the necessary assembly instructions directly at the workers' fingertips by introducing mobile devices fitted with an AR application. But first, we should determine how instructions data have to be organize so it can be served to a worker.

As we have seen before, a ship is formed by several parts (eg. units, sections, blocks), which in turn are formed by several components. As such, these parts can also be seen as components, formed by other components but still a component. In essence, a task is a task, no matter the component(s) in it. Following this logic, we arrive at what we refer to as a **Work Package**, a bundle of all components necessary for a specific task that a worker must complete, despite the stage in the construction.

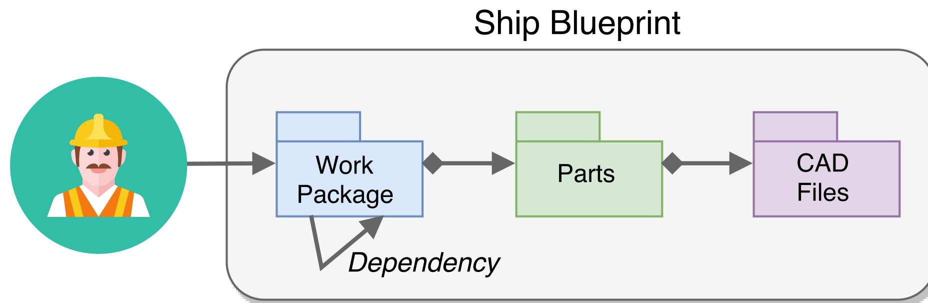


Figure 2.10: Work Package, a bundle of parts, where each part is represented by one or multiple corresponding CAD file(s). Work Packages have a dependency relationships with other Work Packages.

To put in context, the unit depicted in Figure 2.6 can be defined as a Work Package, the five sub assemblies other Work Packages, which in turn are comprised of several other components, as we saw in Figure 2.5, and so on for the minor assemblies seen in that figure.
Figure 2.10 illustrates that logic, where a work package is a bundle of parts, and each part is represented by one or multiple corresponding CAD file(s). Since we establish that some components/tasks are connected or are dependent on others, that is, a ship is dependent on its blocks, every block is dependent on the sections that comprise it, and so on, and since any of these are a Work Package, there is a dependency association between some Work Packages.
To understand the Work Packages dependency structure on a larger scale, Figure 2.11 illustrates that organization, where it is possible to see the individual dependency relationships of the Work Packages.
Next, we must consider how work packages are categorized within this structure, in other words, how can we differentiate between a work package belonging to a ship, a particular stage in the construction, and how these correlate with the workers. All so we can identify which work package we must deliver to be a worker.

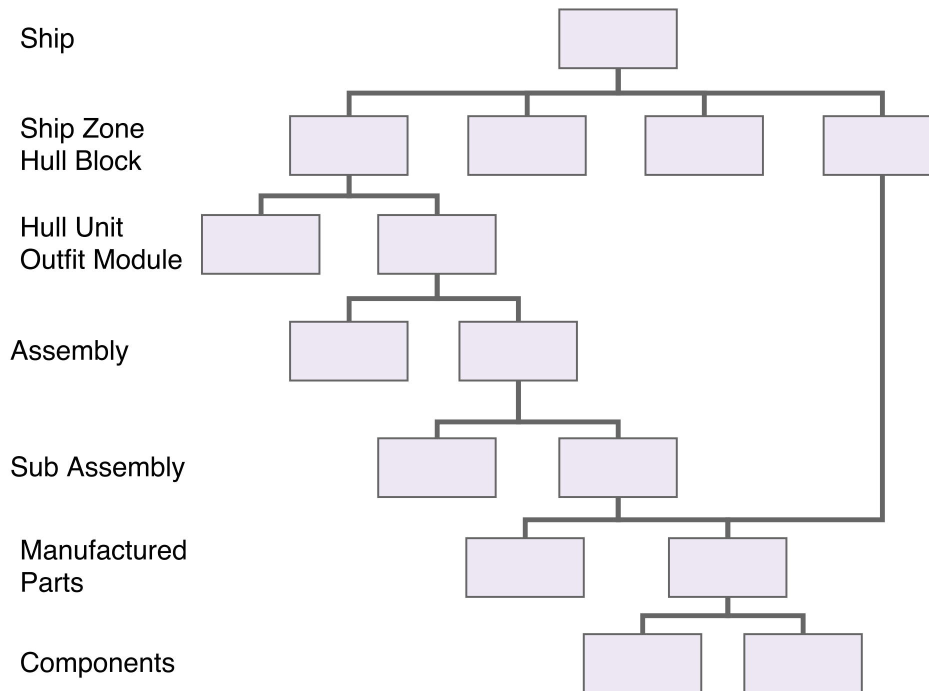


Figure 2.11: Work package dependencies structure.

After great analysis/reasoning, we arrived at the possible categorizations:

SHIP A specific ship or instance of a ship. Eg. ship's name or in-house reference.

DEPARTMENT The stage of the construction. The name department is due to different stages of the construction taking place in different locations, as we have seen in Figure 2.4. Examples of departments can be "Block" or "Section".

TYPE OF CONSTRUCTION The specific type of work to be performed. For example, "Welding" or "Electrical" installation. This is particularly important in case a specific task requires a worker to have an explicit expertize.

STATUS The current status or condition of a Work Package, such as "Complete" or "Delayed". This way, we can seep through the ones that must be completed. Also, we can have a clear view of the percentage of the ship that is completed and if the construction is progressing as scheduled.

WORKER In the cases management wishes to assign specific tasks/Work Packages to particular individuals.

Expanding the work package representation that is depicted in Figure 2.10 by introducing the previous categories, we arrive at the model proposal seen in Figure 2.12. Relation-

ships between categories *(in grey)* and the Work Packages can be seen clearly, including relationships between the categories themselves. Recall that the purpose of this model is to have a clear interrelationship between a worker and a work package, justifying the relation between *Worker* and *Type* and *Department*. For a fine-grained view on the ship's progress, both *Work Package* and *Part* have a *Status*.

Other possible categories such as *Work Period* or *Shift*, and a particular *Work Station* inside a building where the work should take place where also considered. However, in the interest of having a generic/universal model that could be easily adopted by any shipyard, we decided not to enforce those categories into our model. Nevertheless, a certain flexibility is kept in mind, in order to allow these or more categories to be easily introduce in the future.

We believe that this organizational concept model, however simple, can be sufficient to hold all the data pertinent to the assembly instructions that will be later transfered to the prototype, as we will demonstrate in future chapters.

2.4 DECISION SUPPORT SYSTEM

Further to previous comments, presently there are a variety of tools backed by intensive research and algorithms developed for determining the best space allocation for the construction's material, as well as a task plan, i.e. a work schedule for the whole construction phase of a ship. These techniques vary from the fields of project management, chain operations, and intelligent and optimization operation systems. [10] [11] [12] [13]

In the context of our solution, some heuristics have to be set to determine the arrangement of Work Packages. In essence, the definition of a Work Package should not be too far

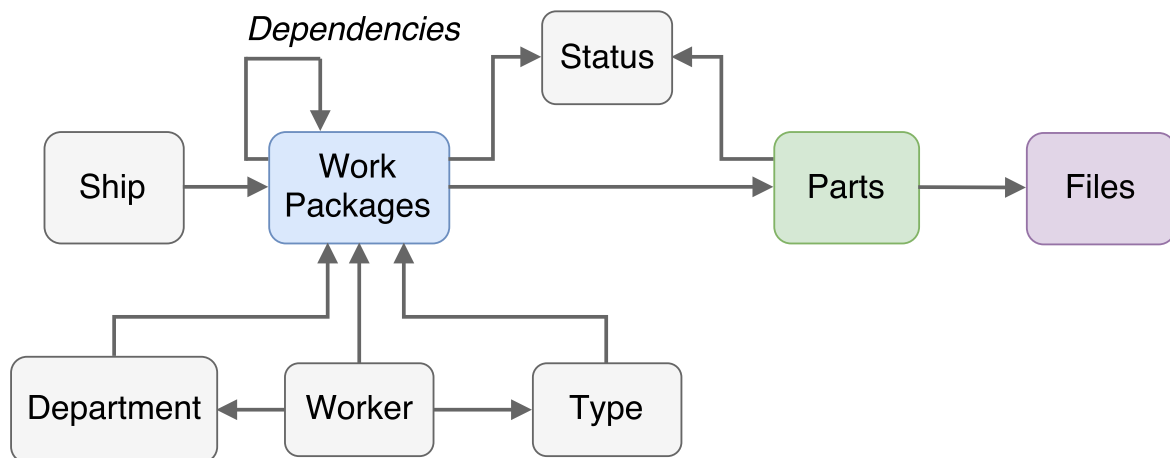


Figure 2.12: Model proposal. A Work Package is a comprised of Parts, which in turn are comprised of Files. Relationships of the categories with the Work Package

from what, even if unknowingly, shipyards are currently using. As we mentioned before, the problem lies in the way this task scheduling and the information associated with the tasks, are distributed to workers.

In the previous section, we described how information about assembly instructions should be formally organized — in correlation with what we identified as a Work Package and a worker. To support this organization, a concept worth studying, would be a decision support system that, after having the complete information about the assembly scheduling and the defined Work Packages for each instance of construction, would form (groups of) workers or/and assign specific Work Packages based on their competences and track record in previously (identical) constructions. Thus, the new construction would be faster and more predictable, accounting even for sporadic absences of personnel and rearranging Work Packages and workers in order to keep the deadlines and budgets.

An unique system, encapsulating the entire construction throughout all the phases and locations, would be too complex and difficult to manage; one small change would have a ripple effect, changing the flow of the construction and the schedule. Adding to this complexity, is the ever present variable of design changes to the initial instructions, forcing the whole schedule and planning process to be repeated. [14] Perhaps for this reason, all the found studies and research, tend to focus on one small assembly stage of the construction, like the outfitting. [15] [16]

Although this thesis' scope does not cover the extensive study of these systems, we will propose some design implementation components with the intention of applying the best solutions, developed in conformity with the topic and which can be added later to an existing or future decision support system. Furthermore, at the end of the thesis, we will describe possible future implementations and enhancements for this work, many of which would fill the criteria of an operation and decision support system.

3 ARCHITECTURE

"When it comes to writing code, the number one most important skill is how to keep a tangle of features from collapsing under the weight of its own complexity."

— James Hague

Chapter 3 pretends to explain the architectural solutions and design plans that, based on the objectives specified in Section 1.2 and the model defined in Chapter 2, were drafted in order to achieve the presented solution. In this chapter we will discuss the planning phase by: **(i)** Analyzing the constraints surrounding the topic; **(ii)** Exploring the different security components such as: encryption, access control, and data protection; **(iii)** By what means the communication and transferring of assembly instructions is carried out; **(iv)** Determining the underlying logic behind the selection of a work package; **(v)** And studying the performance and usability of the overall solution.

Some of the content described here will be of different types: part of it will be applied directly to the final solution; some will be defined and explained as possibles approaches to the project; and some of which will not be applied but is still meaningful and necessary to explain the thought process that lead to the final solution. All of which will be properly identified for better understanding.

3.1 SOLUTION CONSTRAINTS

Unlike their wired counterparts, mobile devices are subjected to severe limitations on power consumption, performance, size, and weight, limiting the range of technologies that we can use. Therefore, we must have a tangible understanding of how every available technology works and how we can benefit from it.

Mainly constraints are imposed by the technologies and features most commonly found in today's mobile devices. In other words, even though there are accessories or attachments that can expand the devices' capabilities, we should limit our solution to what these devices can do on their own. This way, no unexpected costs, that can influence the implementation of the solution we propose, are added to the final plan.

A prerequisite for the solution is that communication between the content provider system and the devices should be performed wirelessly, in order to take full advantage of the devices' portability. The same way, there are also constraints that limit the choice for the wireless technology, as buildings have a great area that must be covered by it. Usually

buildings have around 400 meters in length and are 150 meters wide, the height varies from building to building with some of them being 80 meters tall. The choice of any Wi-Fi technology will come down to a balance between its implementation cost, and the range they can cover, as well as their throughout capacity, not to mention that devices must support this technology.

At the physical layer, where the means of transmitting the actual data exist, there are several protocols. The most popular and commonly found in mobile devices are:

INFRARED Infrared (IR) is an electromagnetic energy not visible to the human eye, i.e., out of the visible spectrum. There are several potential applications, such as night vision and thermography (discover the temperature of an object), but IR can also be employed in short-range communications. These signals have a very short range of around 5 meters, working only in direct line-of-sight, and cannot penetrate walls or other obstacles. This directionality can be seen as an advantage in terms of security, ensuring that data is not leaked or spilled to other devices. The fact that no special or proprietary hardware is required and it can operate at a low power consumption, makes this technology still useful in many of today's electronic equipments. [17]

On the other hand, the data rate transmission is lower than typical wired transmissions, with speeds ranging between 115 Kbps and 4 Mbps, depending on the standard enforced. [18] Research has been conducted to increase these speeds up to 3 Gbps [19].

BLUETOOTH Created in 1994 as a wireless alternative to data cables, it uses radio signals on the 2.4 Ghz frequency band to transmit information. It is relatively easy to use, and has a minimal installation cost, this is why it is widely present in most devices. It is mainly used for transferring sound data (bluetooth headset) or files between two devices that are near one another, in low-bandwidth situations. [20] Bluetooth Low Energy (BLE) hit the market in 2011 as Bluetooth 4.0 which remains in sleep mode unless a connection is initiated, prolonging battery life expectancy.

Bluetooth has a relatively low data rate of 1.5 Mbps and a very short range of approximately 10 meters. Depending on the class, the values can increase to a maximum of 24 Mbits and up to 100 meters respectively [21], however power increase is correlated with increase of range and data rate and not all devices can support those classes. [22]

WI-FI An Institute of Electrical and Electronics Engineers (IEEE) radio standard that provides network access within a limited range, using the 2.4 GHz and 5 Ghz radio band. The Wi-Fi Alliance defines Wi-Fi as any Wireless Local Area Network (WLAN) product based on the IEEE 802.11 standards. [23] Over time, the terms "Wi-Fi" and "WLAN" have become synonymous of one another, since most modern WLANs are based on these standards. It is widely used in today's world, with different applications like providing Internet access and point-to-point communication.

The 802.11 family has several specifications, of which the following are the most commonly supported: 802.11a, 802.11b, 802.11g, 802.11n, and 802.11ac, with the latter only present in most recent devices. There are several differences between these specifications which: range from the frequency in which they operate, data transfer rates, and installation cost.

Table 3.1: 802.11 Specifications Comparison.

Standard	Frequency Band	Max. Bandwidth	Max. Data Rate
802.11b	2.4 Ghz	20 Mhz	11 Mbps
802.11a	5 Ghz	20 Mhz	54 Mbps
802.11g	2.4 Ghz	20 Mhz	54 Mbps
802.11n	2.4, 5 Ghz	20, 40 Mhz	600 Mbps
802.11ac	5 Ghz	160 Mhz	6.93 Gbps

Since it is possible to use multiple antennas, the data rates described in Table 3.1 can be even higher. For example, the theoretical maximum speed of 802.11ac is up to eight 160 MHz 256-QAM modulation channels, each one capable of 600 Mbps; this means we can have a total of 4800 Mbps or 4.8 Gbps. While in real-world scenarios these speeds may never be achieved, we can easily expect speeds of 2 Gbps. [24] For a more detailed analysis of the specifications, the articles [25] and [26] provide a nice overview on the subject. Summary tables of the main characteristics can also be found in [27] and [28].

CELULAR DATA Cellular networks or Wireless Wide Area Networks (WWAN) are designed for wide area coverage transmissions. They are often divided into 2nd Generation (2G), 3rd Generation (3G) and 4th Generation (4G) networks. Typical 2G networks include *GSM*, *EDGE*, and *GPRS*, and were originally voice centric. [29] *CDMA2000* and *UMTS* are standards branded as 3G, with a theoretical minimum throughput of 21 Mbits and a theoretical maximum of 672 Mbits, although typical throughput is around 6.1 Mbits. [30] 4G is the fourth generation of mobile communications standards, the current systems that are deployed widely are *HSPA+*, *WIMAX* and *LTE*. [31] [29] 4G is a successor of 3G and provides even higher bit rates of 100 Mbits and 300 Mbits (theoretically). Technological advances have been made to increase the bit rate to 1 Gbits [31], although the typical throughput, meaning what users experienced most of the time when well within the usable range, is only around 15.1 Mbits [30]

The wide coverage area of these networks is definitely attractive. However, these cellphone systems are primarily owned by telecommunication providers and most of

the today's devices do not come with cellular antennas as standardized equipment, an addition that would require an even greater investment.

Wireless Personal Area Network (WPAN) networks like Infrared and Bluetooth have a cheap implementation cost and an excellent low power consumption, which is a key factor when working with mobile devices. However, their comparatively slow data rate and very short range makes them an unsuitable approach to our problem. On the other hand, WWAN networks, like 3G and 4G, can have a vast area coverage and fast data rates. Nevertheless these networks need supporting equipment like base stations, which involves heavy capital to setup the network and a substantial operating cost to maintain it. Combined with the extra cost of having cellular antennas in the devices can make this approach exceedingly expensive. By critically analyzing the pros and cons of WLAN networks, we came to the conclusion this might be the most suitable choice for the problem we are solving. Wi-Fi is a standard technology used to communicate in a **free unlicensed spectrum**. [32] It has a fairly high coverage radius and signal boosters or repeaters can be easily introduced and set up at low costs. Wi-Fi antennas are standard equipment in any mobile devices like tablets, and the Wi-Fi data rates are steadily increasing, thanks to standards like 802.11n and 802.11ac. The latter promises data rates close to 2 Gbps, which is comparable to rates in 4G networks. Research and trials have also proved Wi-Fi to have better battery life performance than cellular networks like 4G LTE for as much as 25%. [33] [34]

No matter the type, there will always be challenges when dealing with wireless transmissions. Radio signals attenuate over space, i.e., signal weakens as it gets further away from the transmitter due to the inverse square law [35]. Reflections, obstacles and other conditions can also contribute to this attenuation. Furthermore, other radio frequencies, or electromagnetic devices or cable can cause interference with the signal. These are serious concerns considering that shipyards have an abundance of metals and other machineries that have a high interference ratio. [36]

In the following sections, we will describe, along with the architectural plans, the necessary precautions to consider when dealing with problems resulting from network inaccessibility and data losses due to interference.

3.2 DATA TRANSMISSION

As stated before in Section 1.1, all files pertinent to a task have to be manually gathered and transfered to the mobile devices. This is, of course, an ineffective solution for the present goals. It stands to reason that some form of content provider — a *backend* server — should be introduced in order to facilitate the automatic selection and transfer of files (work packages) to the mobile devices. This is what is normally called a **client-server relationship/architecture**, where the mobile devices will be clients communicating (wirelessly) with a server that provides them with the necessary information and any other service.

In Section 3.1, we have established that Wi-Fi constitutes the best overall option as the technology for data transmission. In this section, we will discuss what technologies/protocols are the most suitable for the establishment of the connection between the server and the clients (mobile devices), the type of connection, and how the assembly instructions and other types of data are exchanged.

3.2.1 *Transport Protocol*

As highlighted previously, there are some physical challenges within a shipyard environment: the size of the buildings, the abundant presence of metal and machinery that can cause signal-loss and interference during transmission, which can cause packet loss and data corruption [37].

On top of the physical layer, there are protocols used for transferring data from a source to a destination *via* a network. The more well known protocols in this layer, and also the more supported ones, are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). The main difference between them is that TCP is a connection-oriented protocol and UDP is a connectionless protocol. The former establishes a logical connection between parties before data is sent. With the latter, data is sent without first creating a connection. [38] This difference means that data sent by TCP is guaranteed to remain intact, and in the same order in which it was sent. With UDP, there is no guarantee that the data sent will reach the destination. There are of course drawbacks to having this reliability and ordered delivery. TCP is a much slower protocol and has a bigger header size (20 bytes to 8 bytes) than UDP, which can perform faster due to not having error-checking. These and more characteristics of these protocols can be seen in Table 3.2.

Table 3.2: TCP and UDP comparison.

	TCP	UDP
Acronym	Transmission Control Protocol	User Datagram Protocol
Connection	Connection-oriented	Connectionless
Packet Entity	Datagram	Segment
Header Size	20 bytes	8 bytes
Speed	Slower than UDP	Faster due to no error-checking
Congestion Control	Yes	No
Flow Control	Yes	No
Ordering of data packets	TCP rearranges data packets in the order specified.	No inherent order, all packets are independent of each other. Packet order can be managed only by the application layer.
Reliability	Yes. Guarantee that the data transferred remains intact and arrives in the same order in which it was sent.	There is no guarantee that the messages or packets sent reach the destination.
Streaming of data	Data is read as a byte stream, no distinguishing indications are transmitted to signal message (segment) boundaries.	Packets are sent individually. Packets have definite boundaries which are respected upon receipt.
Weight	Heavy-weight, requires three packets to set up a connection, before any user data can be sent.	Lightweight, due to no message order and no connection tracking.
Handshake	SYN, SYN-ACK, ACK	No handshake (connectionless protocol)
Acknowledge	Acknowledgement segments	No Acknowledgment
Common Header Fields	Source port, Destination port, Check Sum	Source port, Destination port, Check Sum
Use by	HTTP, HTTPs, FTP, SMTP, Telnet	DNS, DHCP, TFTP, SNMP, RIP, VOIP.
Main Usage	Applications that require high reliability, and transmission time is relatively less critical.	Applications that need fast, efficient transmission. Stateless nature is also useful to answer small queries from huge numbers of clients.

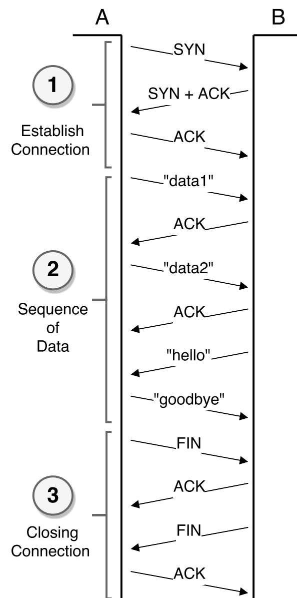


Figure 3.1: TCP sequence diagram, showing the establishment of the connection, the transferring of data, and the closing of the connection.

	Party A	Party B
1	SYN, **seq=0**	
2		SYN+ACK, seq=0, **ack=1**
3	ACK, **seq=1**, ack=1 (ACK of SYN)	
4	"data1", **seq=1**, ack=1	
5		ACK, seq=1, **ack=4**
6	"data2", **seq=4**, ack=1	
7		seq=1, **ack=8**, "hello"
8	**seq=8**, ack = 6 "goodbye"	
9	**seq=21**, ack=6, FIN	seq=6, **ack=21** ACK of "goodbye"
10		seq=6, **ack=22** ACK of FIN
11		seq=6, **ack=22**, FIN
12	**seq=22**, ack=7 ACK of FIN	

Table 3.3: TCP acknowledge and sequence packages, on bold are the numbers used to maintain order and an error free transmission.

Based on the characteristics of these protocols and the physical challenges listed before, we feel TCP is the most adequate protocol for the transport of data between server and clients.

TCP will be responsible for breaking data down into small packages before they are sent over the network, and for assembling the packages again when they arrive. It is built normally upon the Internet Protocol (IP), and provides "reliable, ordered, and error-checked delivery of a stream of octets"[3] [38] between applications running on hosts communicating over an IP network .

Figure 3.1 shows the steps involved in establishing a connection and transmitting data:

1. A connection is established via a three-way handshake. Party *A* establishes the connection by sending a `SYN` request to party *B*. In response, party *B* replies with `SYN-ACK`. Finally, party *A* sends an `ACK`, acknowledging the reply. At this point, both parties have acknowledged and established the connection.

3 An octet is unit of measure in computing that consists of eight bits.

2. Now that the connection is established, sequences of data can be send from one party to the other. Sequence and acknowledgment numbers are used to make sure all packages are delivered in order and error-free. In Table 3.3 we can see how TCP ensures that all packets arrive at the destination, by using `SEQ` and `ACK` numbers.

3. To terminate a connection, a similar situation to the establishment of the session occurs, however this time involving `FIN` packets and a four-way handshake with each party terminating independently. Each party sends a `FIN` packet to the other, and both acknowledge the termination by replying with an `ACK`. Thus, a typical tear-down requires a pair of `FIN` and `ACK` segments from each TCP endpoint [38].

3.2.2 *Transferring Protocol*

Up to this point, we have defined the wireless transmission technology — Wi-Fi — that will be used for communications between server and clients. Moreover, TCP, which is built on top of the IP protocol, will be responsible for establishing the connection and providing the reliability needed for the transmission. As of now, we have the necessary means to send information. Nevertheless, there are still protocols that sit (in the stack) on top of TCP. This is because, although TCP is a "reliable ordered byte stream protocol" [39], information is sent in binary format and with no boundaries between stream segments (Table 3.2).

Hypertext Transfer Protocol (HTTP) is an application layer protocol, which uses an underlying TCP connection (Section 3.2.1) to distribute hypermedia information [4], declaration and negotiation of data representation. [41] While TCP is responsible for the communication establishment, HTTP handles the request itself, stating the message format and transmission type by using error codes and headers, and the action that should be performed. For these reasons, HTTP has since become the underlying protocol used by the whole World Wide Web (WWW) [5]. A more comprehensive explanation of the HTTP protocol, its methods, and headers will be given throughout this thesis.

There are many applications that can make use of the abstraction provided by TCP, and since there are no additional data overheads, these applications tend to have a better performance when compared to HTTP applications. However, dealing with binary streams means that clients must be aware of the format and size, of the data they will receive, which can make TCP applications fairly complex to build. The nature of HTTP allows for systems to be built independently of the data being transfered [41], giving them flexibility, which is why we will be relying on it.

HTTP is what is known as a *request-response protocol*, meaning clients send a request to the server, and when the request is received (and processed), the server sends the response.

4 "Hypermedia is the generalization of hypertext to include other kinds of media: images, audio clips and video clips are typically supported in addition to text". [40]

5 A bit of history: Developed at CERN, the WWW primary goal was to allow hypertext documents to be electronically linked, so selecting a reference in one document to a second one would cause it to be retrieved. HTTP was the mechanism created to so that a client computer could tell a server to send it a document. [38]

This is also known as a half-duplex operation (Appendix 2). The client-server model was originally developed around the idea that the client is always the one initiating transactions, the party requesting information. Up to this point, this kind of model should be adequate and fulfill our requirements. However, and as stated in previous chapters, the assembly instructions might be altered or modified after being initially sent. This situation could result in workers relying on outdated information, resulting in mistakes during the construction. HTTP does not allow the server to independently send data to the client without a request first. Considering alterations to the ship's design, and consequently modifications in the assembly instructions, can occur at any time, we need to find a way to either push the updated instruction to the respective clients *via* a notification, or the data itself, or having the clients periodically send a request for updates.

While researching this subject, we have come across with several methods of communication that can solve this problem:

POLLING One way to solve this problem/issue/situation is by having the client periodically ask the server if there are any new or updated assembly instructions of the design currently being assembled. This approach can be expensive, in terms of resources, due to the constant traffic, and the server's constant processing of requests, especially if there is a great number of clients constantly making requests to the server. In a more detailed view, the client creates a connection (recursively, at given intervals) with the server, sends the request asking if there is any new information, and gets a response from the server. If the response is empty or indicates that there are no new information, the connection is closed; if there is updated information, the client will make another request, this time for that specific information. This solution is supported by all major clients and programming libraries.

LONG POLLING Similar to the previous solution, a connection to the server is established, however it is kept open (or alive) for some time, but not indefinitely. During this time, the client can receive data from the server, and reconnect periodically, even after the connection is closed, thanks to the timeout set at the beginning of the connection. This solution is also widely supported, and differs from the previous one only in the fact that, if there is no new information, the client will not be receiving an empty response. In this case, the server holds the request and waits for a certain period of time. Only then, if there is no new information, the server sends an empty response to the client. Long polling reduces the amount of data that needs to be sent since the server only sends data if there really is any data to be sent.

SERVER-SENT EVENTS (SSE) Enables efficient server-to-client communication. Under the hood, a client establishes a persistent and long-term connection to the server. Note that, with this method of communication, only the server can send data to the client. This is known as a Simplex channel (Appendix 2). In case the client wants to make a request to the server, it has to use another method or protocol, like Polling. Since

SSE uses HTTP, there is no need for a special protocol or server implementation to be put in place. Compared to Long Polling, the advantage of this method of communication is that data is only sent when there is new data, unburdening the client from reestablishing a new connection and processing "empty" responses.

WEB SOCKETS Web sockets are a new standard that allow for a long-held single TCP connection to be established between the client and server. This connection allows for bi-directional, full-duplex (see Appendix 2) messages to be exchanged in a two-way communication between two parties without relying on multiple HTTP connections. Indeed, a TCP connection is created by the client to the server, and is kept for as long as it is needed (*Persistence Connection*). Closing of the connection can be accomplish from both sides. Initially, the client has to go through the normal handshake process (recall Section 3.2.1, Figure 3.1, and Table 3.3), and if successful, at any time, both the server and the client can exchange data in both directions. Since data can be sent from both directions, this protocol is widely used in many real-time applications. Moreover, most of the overhead and time are spent only once in establishing the connection, resulting in a very low latency connection. Web Sockets are widely supported by most servers and clients [42]. The main differences of this method when compared to the others are that Web Sockets usually have a very different logic approach for the networking and the exchange of information. Thus, applications have to be designed, from the beginning, with this logic in mind.

The use of either one of the methods in the context of mobile devices has the potential to affect their battery life/performance. In many of these methods, a persistent connection must be constantly maintained between the server and the client, which can have drastic impacts on the battery life/performance [43] [44]. On the other hand, constant requests can put pressure on the server and the network, especially when there are no new instructions. To determine the optimal transport, there must be first clear requirements and targets for an application. The secret is to determine the best interval in which updates should occur, a compromise between cost and efficiency. An analysis of the most frequently used update times showed that these updates take place no more/no sooner than once per day, in a real-world scenario. Thus, relying on methods like Web Sockets and Server-Sent Events (SSE) , that use a persistence connection, will only complicate our solution. Since the time between requests can be long, we decided to use the simpler and easier option, Polling. To clarify the decision, we are not advocating Polling is the best overall method of communication. The concept of "real-time" has different perceptions depending on the kind of application. While some applications require constant updates (like a chat application) others can work with timed delays in the order of minutes.

3.3 SECURITY

One important aspect we have to ensure is the overall security of our solution. Particularly, in three scenarios: **(i)** Access control to the server, meaning what type of authentication and authorization ensures only legitimate requests and people/workers can access the system; **(ii)** Protection of assembly instructions (files) while in transit; **(iii)** And guarantee that the files, on the devices (client), are not mishandled. Moreover, how to ensure the protection of sensitive data still present in devices in case of theft, corruption, or recycling.

3.3.1 *Access Control*

"Access control is a security technique that can be used to regulate who or what can view or use resources in a computing environment" [45]. Access control systems perform authorization and authentication identification. While they can appear to be synonymous of one another, they are really two distinct terms. **Authentication** is how one proves to be who they claim; **Authorization** presupposes authentication, it is the step that determines what a person can do once inside the system. Basically, authentication is about who somebody is, and authorization is about what they are allowed to do. [46]

3.3.1.1 *Authentication*

As we stated before, the way to authenticate people accessing the server, is to have them present something that proves who they say they are, in our case, a real employee/worker. There are three types of authentication information: [47]

1. SOMETHING A WORKER KNOWS Credentials, such as a password or Personal Identification Number (PIN). This is the most ubiquitous and simple form of authentication, a correct password or PIN grants access to the system. Since each set of credentials are unique, they also serve to identify the person accessing the system. The major drawback is that it is too easy to lose control of them, by either failing to remember them, imparting them to others, or simply writing them down and others reading them. [48]

2. SOMETHING A WORKER HAS A physical device, such as a smart card or a security token. Security tokens (like *key fobs*) are small hardware devices that provide an extra level of assurance through a method known as *two-factor authentication* — workers have a PIN which gives them access to the smart device, which in turn displays a unique identification number for accessing the server. The PIN prevents unauthorized access in case the device falls into the wrong hands. The generated identification numbers are constantly changing, usually every few minutes. Detriment to this method is that an object must be always lugged at all times, and such object might be stolen and used by someone in mischievous actions. [49] [50]

3. SOMETHING A WORKER IS Biometrics, a physical characteristic such as fingerprints, voice pattern, or retina blood vessels. It usually requires special reading equipment that unfortunately are expensive, not very accurate (at the moment), and cannot be available in most mobile devices.

Authentication methods vary from simple to complex and under some circumstances all three of the previously mentioned methods might be combined in order to ensure a more restrict/secure access to a system. Modern mobile devices might not be equipped with the necessary hardware to implement all three methods, thus we decided that a simple *username* and *password* credential access control is the most appropriate form of authentication in our case. In the context of HTTP this authentication method is usually referred to as *HTTP Authentication*. [51] In every request an authorization header is added, and the credentials: *"username:password"* are encoded in base64 [6].

```
GET               / HTTP/1.1
Host:             xpto.com
Authentication:   Basic Zm9vOmJhcg==
```

Note that even though the credentials are encoded, they are not encrypted. Since the credentials are not encrypted, they constitute a security risk, which justifies the use of a cryptographic protocol like SSL to encrypt the whole communication, usually called *HTTP Digest Authentication*. We will later explain, in more detail, the security of the communication in Section 3.3.2. Moreover, in Section 3.3.4 we will debate the design choices of how a worker can login to the system and how his requests are authenticated.

3.3.1.2 *Authorization*

Generally, authentication is the first step of determining whether a worker can have access to the server. After authentication, the following step is to verify if the worker is authorized to perform the request. There are three basic approaches to authorization:

ROLE-BASED Roles are designed around a person's job function, the permissions to perform certain actions are assigned to specific roles. [52] Just like the structure of a business, a single individual can have multiple roles.

IDENTITY BASED An extension of the role-based authorization, identity models enable manage claims and policies in order to authorize clients. With this approach, it is possible to verify claims contained within the authenticated users' credentials. [53]

RESOURCE BASED Access to individual resources is secured through the use of Access Control Lists (ACLs), a list of permissions attached to each resource stating what type of operation(s) each individual can perform on that resource.

6 Encoding algorithm that transforms arbitrary binary data in ASCII text which provides safe variants of the same data.

3.3.2 *Secure Communication*

When the issue of secure information transmission is at stake, there are two fundamental concepts that must be taken into consideration: confidentiality and integrity [54]. Normally, data exchanged between two parties is sent in plaintext [7], which carries security risks.

One way of mitigating a potential attack would be to use a secure communication protocol to encrypt data in transit. [56] Wi-Fi Protected Access (WPA) and Wi-Fi Protected Access II (WPA 2) are wireless security protocols that aim to provide security *via* data encryption [57] and message integrity check. They have been designed to prevent the altering and resending of data packets. [58] There are two versions for WPA 2: Personal and Enterprise. The former uses a shared key to access the network and, contrary to the latter, does not require an authentication server which provides additional security.

The problem with this method is that communication is only secured between the user and the access point to the network. If at some point an attacker is able to intercept the communication, any sensitive information is compromised and can be later used in a harmful manner [59], using the man-in-the-middle attack, for example. For this reason, it is necessary to choose an appropriate method to protect data all the way to the end, as it is a very dangerous assumption to believe that no-one is listening to the communication between two parties.

Cryptography protocols like the Transport Layer Security (TLS) and its predecessor Secure Sockets Layer (SSL) (both are commonly called SSL) are part of a set of networking protocols, that aim to provide privacy and data integrity in a communication between two parties [60]. "SSL is the secure communication protocol of choice for a large part of the Internet community" [61]. According to the protocol release draft, the SSL protocol provides connection security that has three basic properties: [62]

PRIVACY The connection is private because data is encrypted using a symmetric key. After an initial handshake, [8] an unique symmetric key is generated and negotiated between the server and the client.

IDENTITY AUTHENTICATION Optional, but required for at least one of the parties, usually the server. Here, both server and client rely on asymmetric keys to authenticate one another.

RELIABILITY Each message's integrity is assured by a Message Authentication Code (MAC) that prevents undetected losses or alterations of the data during transmission, guaranteeing a reliable connection.

---

7 In the past, the definition of plaintext only meant message text. [55] It since has expanded to include any data that is transmitted or stored unencrypted.

8 Negotiation that dynamically sets parameters for channel established between two parties before normal communication over the channel begins. [38]

Before the client and the server can begin exchanging application data over TLS, the encrypted tunnel must be negotiated: the client and the server must agree on the version of the TLS protocol, choose the cipher, and if necessary verify the certificates. Unfortunately, this adds latency to all SSL connections, since each of the previous steps must have a packet round trip. A typical SSL handshake can be seen in Figure 3.2. Also, to add to the latency, SSL runs over a reliable TCP connection, meaning that a complete TCP three-way handshake (Figure 3.1) that also takes one full roundtrip, must be made. Fortunately, it is possible to optimize the handshake process with *Session Resumption* [63] and *False Start* [64].

Assuming both sides are able to negotiate a common SSL version and cipher, and the client accepts the certificate sent by the client (client verifies the certificate with a Certificate Authority (CA) [9] or by previous verified certificates stored locally), a symmetric key is generated that is then used for all further communications between the client and the server, within that same session.

Although the encryption process may be computationally expensive, its effect on the overall cost is minor. Reducing response time by a few milliseconds does not outweigh the level of security offered by SSL. Developers today have been developing techniques to reduce the overhead and abbreviate handshakes, by reducing the number of round trips for a full handshake, [66] or simply by upgrading their infrastructures. In the future, the amount of milliseconds SSL adds to connections will only continue to decline. [67] [68]

3.3.3 *Data Protection*

Assembly instructions that are transfered to the devices should not have a long lifespan. Once a worker no longer needs certain instructions, they should be erased completely to prevent foul play. Nonetheless, during that lifespan there is still the possibility that those instructions can be somehow misused. Digital Rights Management (DRM) systems were created, in essence, to ensure that a user can only perform authorized actions to a resource. They are mostly used for digital media content such as audio files. For example, a user can only play a specific audio file that is protected from copying or sharing by DRM. [69] The fundamental problem with DRM is that the content is also distributed with the key that opens it, or a key is granted by accessing an external service/server. Keys are hidden such that only those that know its location can find it, however it only takes one person (or a group of persons) to find where it is stored for the content to be accessed freely, making the protection of other similar contents irrelevant. DRM has been proven to fail multiple times [70] [71] in preventing unauthorized access, most of the times due to poor implementation [72], resulting from gratuitous complexity in order to obscure the key used.

9 A CA is an entity that manages digital certificates — issues and revokes. Digital certificates certifies ownership of a public key. [65]

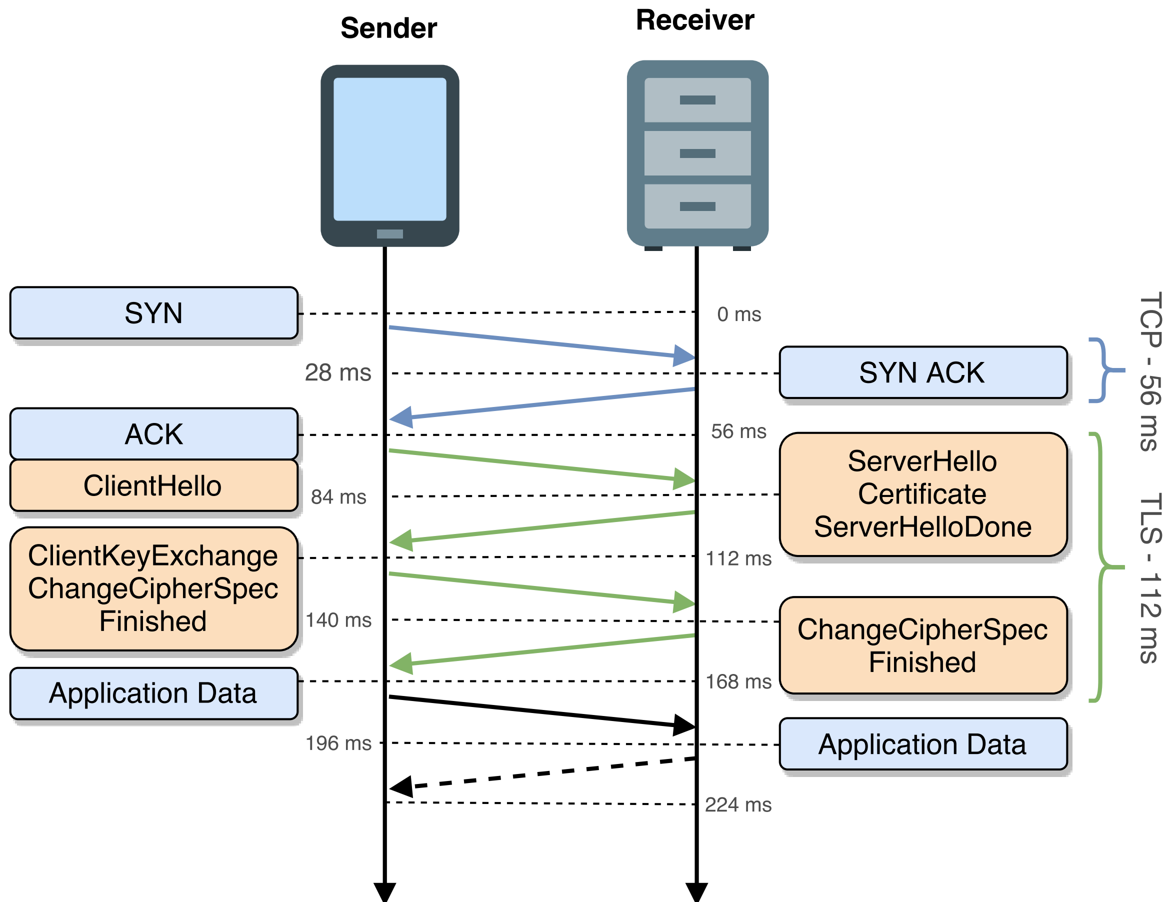


Figure 3.2: SSL connection establishment. TCP + TLS handshakes, and afterwards application data can be exchange on the secure channel.

Even so, we still need to add a layer of security to prevent files mishandle. A solution would be to encrypt [10] the files sent to the devices, which can be later deciphered by a key — content encryption key This means that even if the files are intercepted, moved, or copied, they would be useless without the key to decipher them. However, we must ensure that a key is only used for files transfered to one client and even only on one occasion as an extra security measure, so that it can only be used for those files and no others.

The best option would be to generate the key or use some random data lying around, although the latter could be unsafe. Public key cryptography is a good option, although public keys are slower (when compared to symmetric keys) and they are incredibly more difficult to manage in comparison. In section 3.3.2, we determined SSL to be a good measure to ensure that communication between server and client is private. The SSL symmetric

10 Encryption is the process of transforming data/information so that it is unreadable by anyone who does not possesses a decryption key.

session key, the one agreed between the server and the client, can also be used as the content encryption key used to encrypt the files. Since a new key is generated in each connection it will guarantee the key is only used for a certain transfer and for specific files. Still, to improve performance, a SSL connection can be re-used to spare the handshake process, which we saw, can increase the response time.

In conclusion, generating a separate symmetric key that can be managed more freely is our best choice for a content encryption key.

3.3.4 *Access and Request Authentication*

Regarding the authentication, there is one more architectural design decision we must discuss, the authentication of a user and the requests made by him. As explained in previously, HTTP is a stateless protocol, meaning it does not keep a state between different message exchanges. Stateless can help reduce memory usage in the server, since there is no need to keep track of every user. Problems related to session expiration also decrease, and since there is no session data in the server, there is no need to synchronize data between servers in a distributed environment.

Using HTTP Authentication (Section 3.3.1.1) is a valid choice but it comes with some limitations. Since the username and password are sent in every request, there is no way for a user to login or logout of our system. Moreover, it is a major security risk to be always sending username and password with every request, even when using encrypted channels, we must always assume something might go wrong and the requests can be hijacked.

By using **sessions**, These are difficult requirements to meet when implementing features like authentication and sessions (see Section 3.3). There is often a tight relationship between authenticating users and holding their sessions. A valid session key can authenticate the request and its user, the question is whether this key is stored on the server or an information only the user and the server know. A well known implementation of this logic is the request authentication performed by Amazon Web Services [73], which uses a HTTP scheme based on Hash Message Authentication Code (HMAC) that forms a concatenated string with elements known by the server and the client, encrypts and sends it with every request:

```
digest = base64encode(hmac("sha256", "password", "GET+/"))
```

Since the server has access to all the same information a client does, the request can be validated. Although, to achieve this, the password must be stored unencrypted at the server, which can be a security risk. There are numerous occasions in which user's passwords were leaked due to this practice, for example [74] and [75]. In Section 3.4.2 we will discusses security protective measures for such cases.

The major difference between a regular request hijacking that contains a session token or id, and a hijacking of stateless request is that successful theft of a stateless request

authenticates an attacker even if the victim has logged out (since we do not have the same control as with sessions). In this sense, by measuring the trade-off between computational resources spent decrypting a key (HMAC) and server-side state, we feel that generating and storing a server-side symmetric crypto key we can easily manage and trust is the most efficient way to secure and authenticate every request. Since we have absolute control over the creation of this key and its expiration time, we can use it as the content encryption key discuss in Section 3.3.3.

As such, when logging into the system, through the application, a user inputs his credentials — *username: joe ; password: joepass* — which are then sent to the server:

```
POST        / HTTPS/1.1
Host:       xpto.com
________________________________________
BODY
Username:   joe
Password:   joepass
```

The connection is currently encrypted thanks to SSL. Still, for added protection, we issue this request with a `POST` method, instead of a `GET` method, because the latter exposes information *via* the Uniform Resource Locator (URL). The server then authenticates the credentials and if correct, generates a session key (`1c24171393d4ffcbf11ab28`) for that user. The session is stored on the server, and it is used to validate all subsequent requests:

```
GET        /aService&session=1c24171393d4ffcbf11ab28 HTTPS/1.1
Host:      xpto.com
Session:   1c24171393d4ffcbf11ab28
```

The previous request shows the session key passed in the header of the request or as a URL query parameter (`&session=1c24171393d4ffcbf11ab28`). In our system we accept both, on account of many programming libraries inability to handle HTTP header manipulation, or even the creating of custom headers. A best practice is always to make use of the standard headers defined in the HTTP protocol, however we wanted to take all possibilities into account. There is always more information stored in the database, that can be sent in a response such as the name of the worker and the department. However, only the session is a requisite, while the rest is just complementary information.

3.4 RELATIONAL DATABASE

In Chapter 2 we created a model — Figure 2.12 — that can uphold the information structure necessary to run the current application. Databases allows us to store information in an

organized way. Therefore, we must find a representation that can enlighten the entities constituting our model, and the relationship between each one of them within the database. Likewise, we have to consider the necessary attributes and design choices to guarantee that information can be secured and read as fast as possible.

3.4.1 *Model Representation*

Entity-Relationship (ER) diagrams communicate abstract representations of a data model and the conceptual database design. Represented in Figure 3.3 is the ER diagram of our database. In this diagram we can see the same objects present in the model now represented as entities (rectangles, colored blue). Each entity has a set of attributes (ellipses, colored gray) that defines them, e.g., *Name* and *Description* in the case of a Work Package entity. Most of the attributes are evident, particularly the ones related to names and descriptions. Additionally, we can see some attributes related to time management and progress, these will become clearer in the implementation discussion of the actual database in Chapter 5. Essentially, time related attributes will be used for the verification of updated instructions, and progress related attributes like *Mounted* and *Status* will aid in the selection of the data sent to the devices.

Another aspect that has become clearer is the relationships or associations between the entities (diamonds, colored yellow). An essential element of this diagram is the mapping cardinality, which expresses the number of entities to which another entity can be associated with, e.g. a ship entity is associated with one or more multiple work package entities, 1 – N. [76] These relationships form the basis of how information is organized and how we can search for a particular one.

3.4.2 *Password Protection*

When storing any kind of passwords we have to make a decision about how to store the worker's password information securely. Data in the database is not safe, not only is it accessible by the database administrators, but in case the system is compromised it can lead to passwords being leaked and used in a malicious way. More times than not, we receive news about the theft of large collections of passwords, such as [74] [75], and most recently [77]. Using password scramblers can add some obscurity to the passwords, but this is not considered to be truly secure. [78]

There are two ways in which we can obscure passwords in our database: **hashing** and **encryption**. The former is considered to be the best approach since it is a one-way function where the hashed value cannot be reversed to its original form, i.e, the password. The latter, symmetric encryption, relies on a symmetric key and its always possible to reverse a value by using that key. [79]

So how does our solution authenticate workers with a password hash? As we seen in Section 3.3.4, a request containing the username and password is sent to the server. A

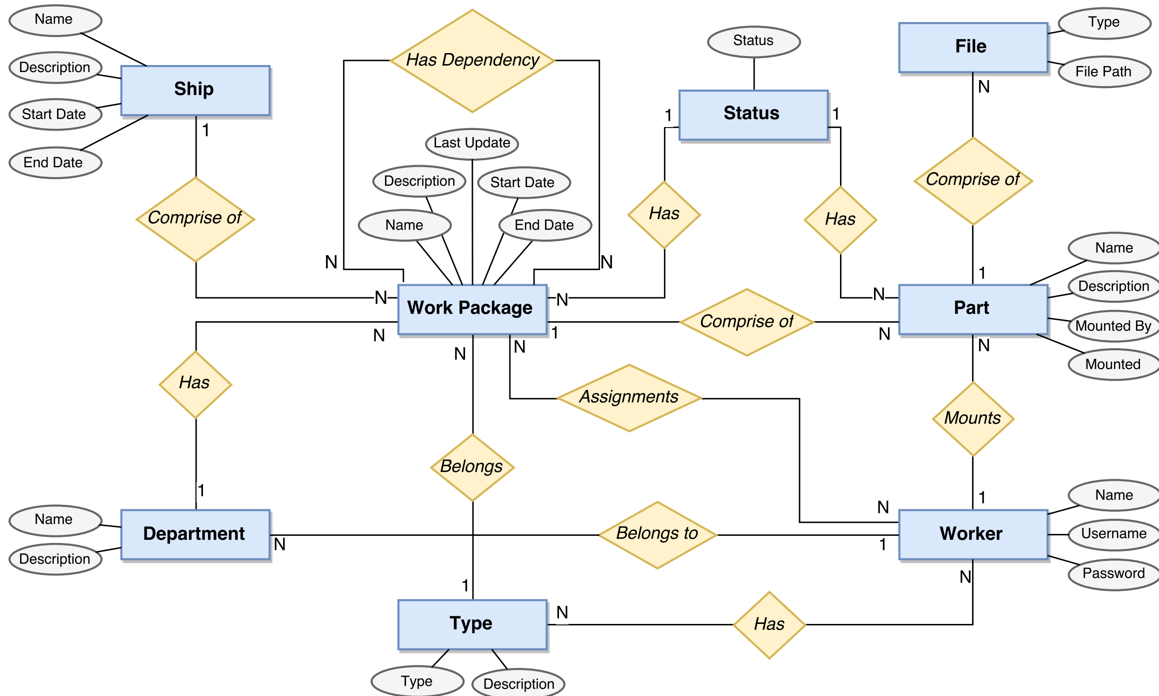


Figure 3.3: Entity-relationship diagram of the model. Rectangles represent entities, ellipses represent attributes, and diamonds represent the relationships between entities.

close examination of the Worker entity in Figure 3.3 reveals the answer we propose. When receiving a request containing a user's credentials, the server hashes the password received and compares it to the hashed password stored in the database, under the username in the request. If the two are an exact match, the worker provided a valid username and password. The benefit of hashing is the application never needs to store the clear password, only its hash value, so even if the passwords are leaked, they are still secure. The process can be seen represented In Figure 3.4.

The Secure Hash Algorithm (SHA) are a family of cryptography hash functions developed by National Institute of Standards and Technology (NIST) [80], and include the algorithms SHA-0, SHA-1, SHA-2, and SHA-3. In terms of interoperability, SHA-1 is the most supported and common of the previous algorithms, present in many programming and environment languages, however SHA-3 has quickly become the new adopted standard which yields a 512-bit hash output and similar performance as SHA-1, which only yields a 160-bit hash output. [80]

There is yet another security measure we can add to secure our passwords, a **Salt**. Lookup tables and rainbow tables [81] are effective methods for cracking password hashes. The reason is that each password is hashed exactly the same way, depending on/under the hashing algorithm, Table 3.4. If two users have the same password, they will have the

same password hashes. [82] In order to prevent this, we can randomize each hash, so that identical passwords will not have the same hash value. We can achieve this by appending a salt value to the passwords before they are hash, Table 3.5. Salts are random, only used once, values and can be stored clear in the database, as can be seen by the ER diagram represented in Figure 3.3.

Table 3.4: Password Hashing. Two identical passwords result in the same hash value.

User	Password	Hash
Joe	password123	le3fh2jka11n...
Susan	password123	le3fh2jka11n...

Table 3.5: Password Hashing with Salt. Appending a salt to a password, before hashing, guarantees that two identical passwords do not result in the same hash value.

User	Password	Salt	Hash
Joe	password123	qw2jknf1...	le3fh2jka11n...
Susan	password123	2awi81nv...	bah1xc91syn...

3.4.3 *Assembly Instructions Storage*

Regarding the storage of the assembly instructions files, two possible design choices exist: storing the files in the filesystem, with the path in the database; or store the files as Binary Large OBjects (BLOBs) in the database. So, the question is to BLOB or not to BLOB? [83]

Databases can manage a number of small objects, and we know that filesystems are incredibly efficient at handling large objects. So where is the division? From what point is reading the file from the database faster than reading a file store in the filesystem? A study concluded that the answer is "BLOBs smaller than 256 KB are more efficiently handled by a database, while a filesystem is more efficient for those greater than 1 MB". [84]

Mayer Shipyard provided us with assembly instructions of a simple section. After analyzing the its size, around 90% of the files' size are bellow the breaking point of 256 KB. However, since the files correspond to a small example of a ship's section we believe that a real ship's section files will surpass the 256 KB breaking point. For these reasons, we decided it is best to store the assembly instruction on the filesystem as opposed of storing them in the database, as BLOBs.

Although the 256 KB breaking point is a good measure to decide between the two design options, there are also other factors that can be taken into account. When storing files' path in the database we must be aware that, in case files are moved, the reference path can break the system. Even though we do not expect that to happen often, it is still worth to

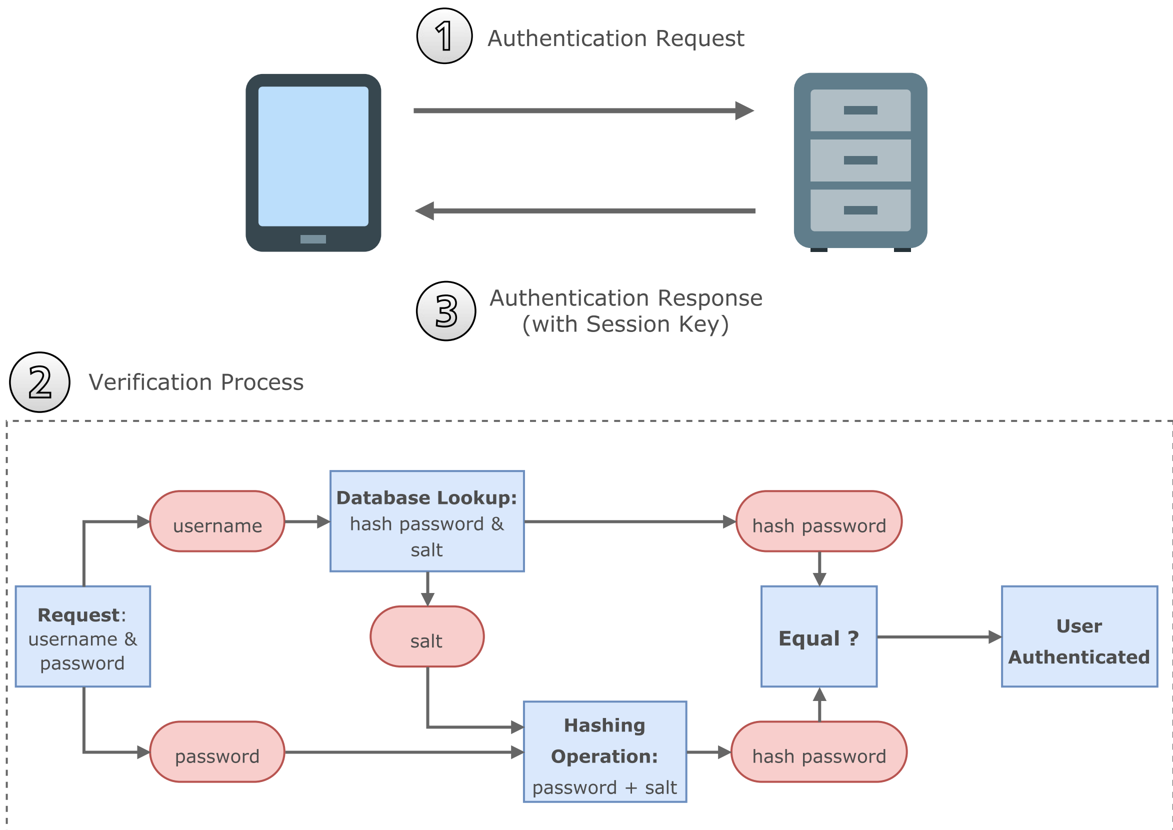


Figure 3.4: Authentication Process.

be taken into account, and the database administrator should be aware of such situations. With such a risk, we would think that storing BLOBs would be the best choice, transferring data to another database would also be easier, since we do not have to account for the risk of breaking files' path. Nevertheless, there are also disadvantages: besides the already mentioned performance when dealing with large objects, if the database is hosted outside the network by a third-entity, the space cost of the database would be immense due to the space required to store the BLOBs; backups of the database, a good measure in case of lost data situations, would also contribute to that expense, due to their size.

When deciding how to storage assembly instructions we believe that storing the path is still the best design choice for our solution. Still, other solutions might not benefit from it, and in the future this decision can be reversed if necessary.

3.5 SYSTEM ABSTRACTION

Described above are the core transmission technologies used to deliver the assembly instructions from a server to the clients, as well as the security measures to ensure a private

and safe communication between them. After a careful analysis of the requirements established for this thesis, we arrived at four core components that we think have an integral role for developing a complete server solution. Figure 3.5 represents an abstract overview of the system and the four components that constitute the server.

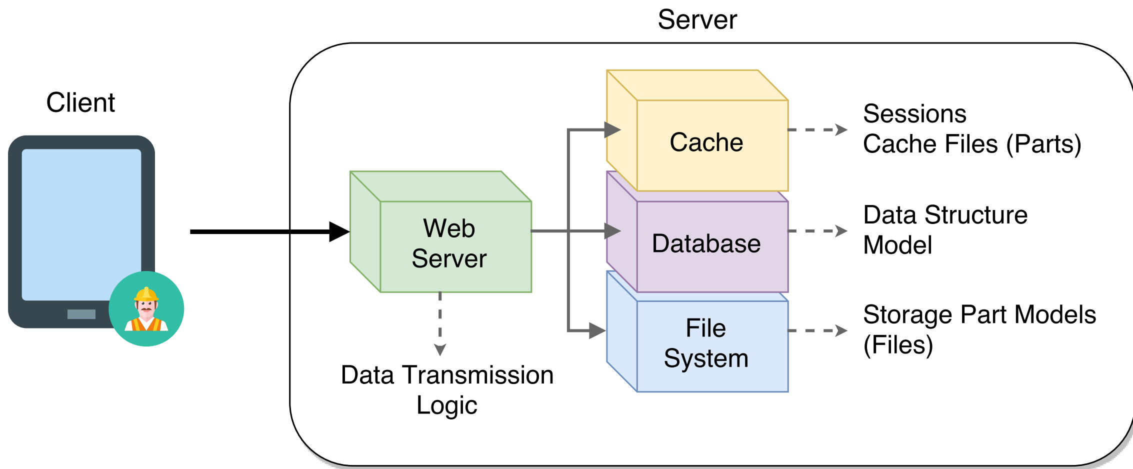


Figure 3.5: System abstract overview.

DISK FILE SYSTEM Storage unit where the various assembly instructions — CAD files — are stored. This system is used to control how data is stored and retrieved.

DATABASE SYTEM We require a database to organize and manage information such as work packages, and their relation with workers and the construction's stage. Information is organized in relation to the model depicted in Figure 2.12. Section 3.4 describes/explains the steps employed in implementing our model as a (relational) database;

CACHE This component will store data in memory, so that future or repeated requests can be served faster; in our case the data stored in cache is a duplicate of the data stored on the disk. Additionally, certain messages/responses can also be stored if needed repeatedly. Cache memory can have reads up to 80 times faster compared to disk reads, [85] thus by storing data in memory, as opposed to repeatedly reading it from the disk, it will help improve the system response time. [86] This component will also be responsible for handling access keys necessary for access control, Section 3.3.

APPLICATION SERVER This component resides in the middle of the server-centric architecture, in charge of handling all application operations between clients and other *backend* applications, services, and databases. It is also responsible for handling connections to the database and other services on one side, and connections to the clients on the other. Services are made available to the client, through a component

Application Program Interface (API), which represents the business logic — a set of operations based on rules or workflows that dictate how data/information is handled, modified, stored, and displayed.

In the upcoming/next/impending sections, we will describe how these components are interconnected, and the diverse array of *middleware* services implemented for security and state maintenance, access control, as well as data access, persistence, and transmission.

3.6 ASSEMBLY INSTRUCTION TRANSFER

Up to this point, we have found that Wi-Fi and HTTP, which uses a TCP connection, are the best transport and transfer protocols/technologies for our solution. Also, by using SSL, the connection between server and client can be encrypted and is guaranteed to be private. In this section, we will be covering how data, pertinent to the Work Package and their parts (the CAD files), is sent to the devices, using the technologies described previously.

Both at the server and at the client, there are many ways in which business logic can influence how we use the technologies at our disposal. For the better part of this section, we will be mentioning the reasons why we decided to follow one path instead of the other. It is consider that, from this point on, a request made by a worker has already gone through the authentication process and has been clear for accessing the system.

The body of every HTTP message is defined in the header field — `Content-Type` — so that the receiving party knows how to process the data in an appropriate manner. To better understand these messages, specially text data messages, there are formats that help to give messages, a structure so they can be easily parsed by programming languages and become human readable. Among the data format interfaces, the most famous ones are Extensible Markup Language (XML) [87] and JavaScript Object Notation (JSON) [88]. The main difference between the two is that XML is, as the name indicates, a markup language, whereas JSON is a way to freely represent objects. Since XML must have a defined data structure it can be validated before being transmitted, however its validation and parsing can take a considerable amount of memory and computer power. The strong integration of JSON into the WWW ecosystem, due to its native way of representing javascript object trees, has lead to its rise in popularity, with many big companies adopting the format. This is also due to its light size when compared to XML, and also the less verbosity, being able to represent the same data with less characters. [89] [90]

For these reasons, we decided to use JSON as the message format for messages exchanged between the server and the client. JSON is built on two data structures: a set of name/value pairs (objects) and a list of values (array or list), universal data structures in almost every programming language. [88]

3.6.1 *Work Package List*

In order for the worker to choose the appropriate Work Package or task, we require a list of possible Work Packages that are available to him. An argument can be made that a worker should not have any input on what work he wishes to perform, but instead, it should be assigned to him. A justifiable decision specially when the device running the application is an OHMD, where there are fewer and more complicated controls to interact with the device. However, we believe this restriction should not be done in the server, but rather, if necessary, on the client application. The application can display the list of possibilities to the worker and let him decide, or can simply decide by itself based on the attributes of each Work Package — the first on the list or the most urgent one by inspecting the end date.

A restriction on the availability and distribution of Work Packages, at the server level, would generate a chokehold on the information available to the client application and hamper future modifications, rising the new of two separate systems.

The Work Packages presented in the list sent to the application must obey a set of rules:

- They belong to the same **Department** as the worker;
- They have the same **Type**, meaning the worker has the necessary skill set to assemble those Work Packages;
- **OR** it can be any Work Package that was **directly assigned** to the worker, regardless of Type or Department, although assigning a Worker to a Task should not break the above two rules, it is possible and can give more flexibility to the distribution of workers in certain cases;
- Another rule, which does not break the previous ones, is the restriction by **Status**, for example including only Work Packages that have a specific Status like "Not Completed". This rule will depend on the Status defined when implementing this solution.

Every Work Package presented in the list should have all the corresponding attributes linked to it by the relationships defined in Figure 3.3. An example of this list can be found in Listing 1. As mentioned before, the client application can choose to display this list or make a selection without the input of the Worker. Attributes in the list can help Workers filter or sort the list to aid in the selection process.

Listing 1: List Work Package Structure. In the list we can see all the Work Packages belonging to a Ship, that match the set of rules for the creation of the list, together with all the attributes related to the Work Packages.


```
[ {
  "ship": 1,
  "ship_name": "Blue1",
  "ship_description": "No Blueprint Description",
  "ship_start_date": null,
  "ship_end_date": null,
  "work_packages": [ {
      "id": 1,
      "name": "WP1",
      "description": "No WP Description",
      "status": "Not Complete",
      "start_date": "null",
      "end_date": null,
      "department": "General",
     "assignment": "true"
    }, {
      "id": 2,
      "name": "WP2",
      "description": "No WP Description 2",
      "status": "Not Complete",
      "start_date": "null",
      "end_date": null,
      "department": "General",
      "assignment": "false"
    } ] } ]
```


3.6.2 *Data Selection*

In Chapter 2, we explained how the construction of a ship takes place. We also introduced our concept of Work Packages and how it translates to hierarchic dependencies matching the modular block construction seen in modern assembly processes. Alongside with it, we proposed our model and how information related to the construction is associated with a Work Package. Later, this model was transformed into an ER diagram (Figure 3.3), which corresponds to the database design that will store the information. As mentioned before, shipyards have several tools [91] at their disposal to discover the best assembly schedule and the material's space allocation. As a rule, blocks are assembled in a succession order, starting from the stern to the bow, Figure 3.6.

To understand the selection of the Work Package described henceforth, we designed a simple fictional example of a ship's block that can be seen in Figure 3.7. The block is divided into two sections, each containing several units, that can be formed by numerous parts like in Unit 5 (U5). As we established, some Work Packages have dependencies on

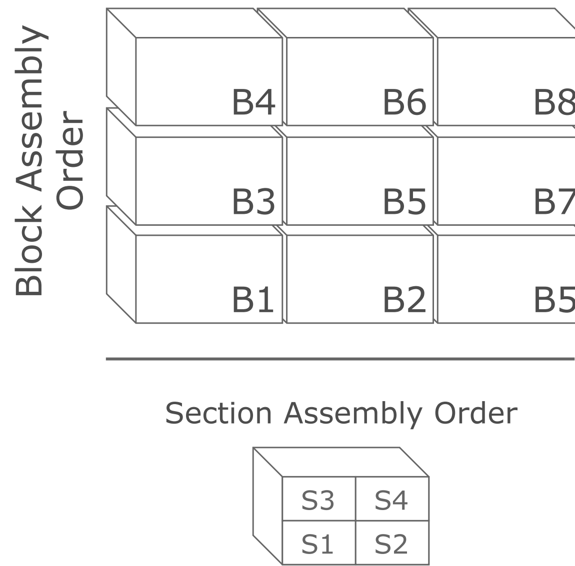


Figure 3.6: Block and Section Assembly Order

others; for example Unit 5 (U5) has dependencies on four Sub-Assemblies, with every one of them containing four parts. There are other relationships as well, Units 3 and 4 have dependency on Units 2 and 1, respectively. The direction of the relationships is a direct inversion to the assembly order, and can be seen clearly in Figure 3.8. Since sometimes it is not clear how to set up these relationships, we also accounted for bi-directional relationships between Work Packages. Even though these relationships may seem close to a tree data structure, they are more close to a graph structure and there will be cases in which we cannot allow relational associations or cyclic dependencies. In Chapter 5 we will cover this subject in more detail, and enumerate how we dealt with such scenarios.

We hope that with these examples (Figures 3.7 and 3.8) we can convey our solution in a more clear way. We will be using these illustrations throughout the remaining parts of this thesis. In the last page of this manuscript, the reader can unfold a page containing the same figures, and use it to understand parts of the thesis, as we describe and make reference to part of the illustrations.

The ship's designs, although detailed, do not cover every single assembly step related to the construction. Most of the times, workers have to make decisions "on site" that are not explicitly expressed in the drawings. So, depending on the type of work they are performing, workers may need information that is not always related to the task at hand. In this section we introduce three methods for Work Package selection. Each method is an augmentation of the previous, and will therefore also carry more information. The reason for all three is based on those situations where a worker may or may not need more information that is not always related to the task at hand.

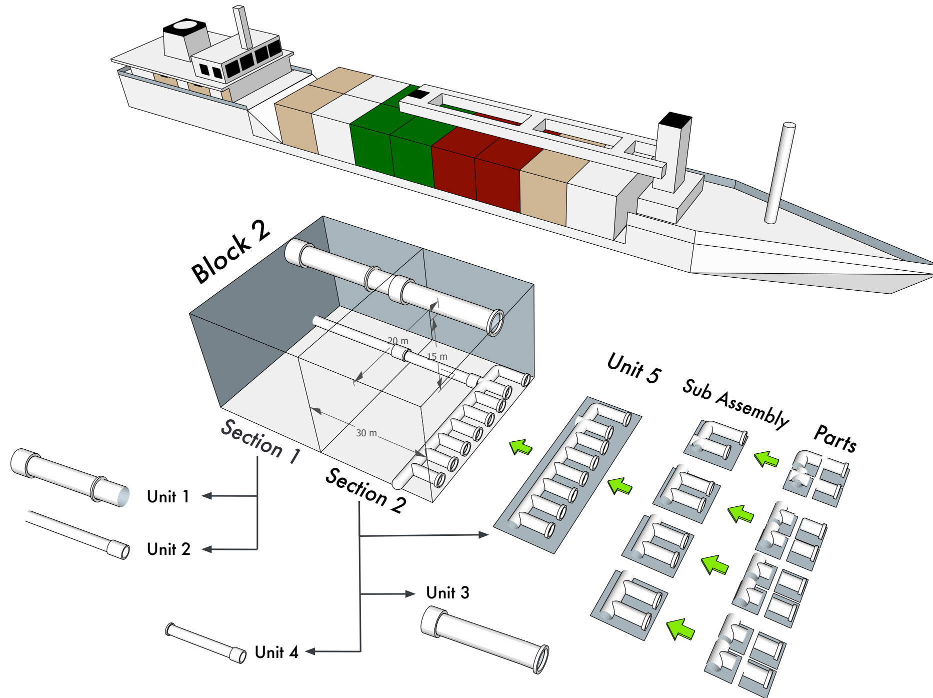


Figure 3.7: Ship Example. A ship's block, which contains a section with 2 units, and another with two units.

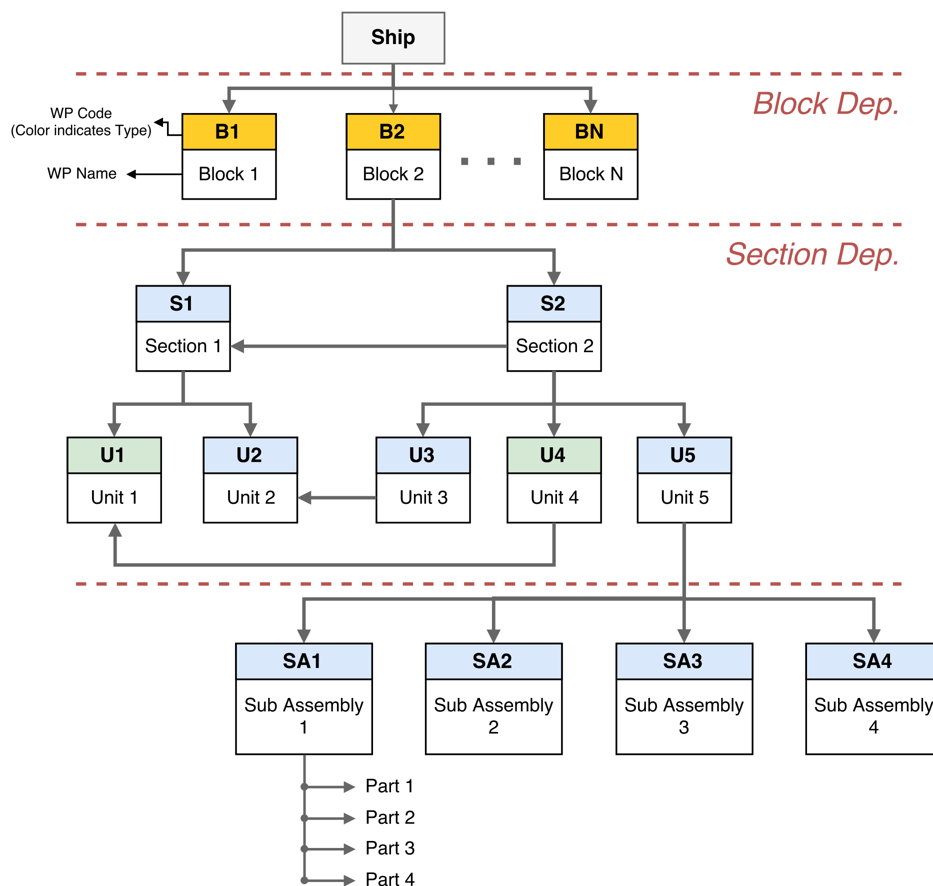


Figure 3.8: Work Package structure, in accordance with the components seen in Figure 3.7.

For all methods, we will be using **Work Package U5** as the Work Package selected (Section 3.6.1) for transmission. As a rule, when a worker is working on a Work Package that has dependencies on other Work Packages, those Work Packages should have been assembled previously, although this may not always be the case.

The three methods are as follows:

SINGLE SELECTION The first method is called simple or single selection. As the name indicates, only one Work Package — U5 — and no others will be sent to the device. There may be occasions in which a worker only needs to see the parts necessary his task, and has no need for parts in other Work Packages, even if they are related to it. Even though use cases for this selection may be scarce/rare/singular, we felt the choice should be available if ever these cases become more frequent.

DEPENDENCY SELECTION Dependency selection is the most discernible of the methods, due to the dependency relationship of the Work Packages. In the case of U5, it means that U5 and all the Sub-Assemblies will be selected. If the Work Package requested was U4, U1 and all potential/possible Work Packages that U4 had dependencies with, would be selected. The ability to see all the components related to the task at hand will allow the workers to see if any previously assembled parts were assembled correctly, and in essence aid in the understanding and performance of his task.

ASSOCIATED SELECTION The last selection is called associated selection. With this, a worker will be able to see all the Work Package associated with his task, independent of it having direct dependencies with other Work Packages that are built in the area/-location/department he is working. U5 belongs to Section 2 (S2) of our block, thus in an associated selection, S2 and all their dependencies will be selected, meaning U1, U2, U3, U4 and U5. This is particularly useful to give an idea of the full context in which the task currently being performed will fit in the overall design of the ship. Moreover, small building decisions, such as where to place small support beams or fittings, are being decided by workers on the stop. By having knowledge of where components will be placed, a worker can place these support components as to not obstruct access to areas he or other workers might require.

The challenge is in evaluating when to use each selection method. In practice, associated selection can be the most commonly used, and could have been the only method provided, as it can convey at least the same information as the previous two. The reason for having three methods is: first for security reasons, as not essential information should not be transfered; and so that the transferring time can be as short as possible, particularly since multiple workers will be simultaneous asking for information, eg. at the beginning of the work day or shift.

Through the Work Package List — Section 3.6.1 — the application has access to a list of Work Packages available for transferring. Above, we explained how all pertinent data to

the selected Work Package, depending on the method, is selected. A request containing the desired Work Package and selection method is made from the application to the server:

```
GET        /workpackage/u5/dependencies HTTPS/1.1
Host:      xpto.com
Session:   1c24171393d4ffcbf11ab28
```

Upon passing the authentication process, the Work Package `U5` is passed through the `Dependencies Method`. A JSON file, containing a detailed description of the Work Packages, their parts, and files is then generated and sent to the application. In Section 3.6.3, we will see how this file is used in the transfer. Listing 2 is an example of the file generated in response to the previous request, the Work Packages in the file correspond to the ones featured in Figure 3.8. An array of Work Packages resulting from the `Dependency Selection` of U5 — U5, SA1, SA2, SA3, and SA4 — and each contains the information associated with a Work Package. The dependencies each Work Package has with others are also present, for example, in Line 4. As a result of the `Dependency Selection`, all Work Packages of `U5` and all Work Packages of U5 dependencies and so on, should be present in Listing 2 — List of Parts.

Following the relationships set in the database design, Figure 3.3, each Work Package is comprised of one or more parts, each of these parts is represented by one or more files. That structure can be easily recognized through the indentations in the file, although these indentations are usually used to make such files human readable. In order to decrease the size of these files, and in turn increase the transfer speed, the whitespaces are eliminated forming a single line, which computers can still efficiently parse.

Listing 2: List of Parts for U5 (requested) on dependecy selection. Small attributes were eliminated for brevity


```
[{
   "id": 1,
   "name": "U5",
   "dependecies": [ 2, 3, 4, 5 ],
 },
 {
   "id": 2,
   "name": "Sub Assembly 1",
   "description": "WP SA1 Description",
   "start_date": null,
   "end_date": null,
   "department": "Section",
   "status": "Completed",
   "dependecies": [],
   "parts": [
     {
       "id": 134345,
       "name": "Part 1",
       "description": "No Description",
       "status": "Completed",
       "files": [
         {
           "id": 1,
           "extension": "txt",
           "path": "/SA1/p1/file1.txt"
         },
         {
           "id": 2,
           "extension": "obj",
           "path": "/SA1/p1/file2.obj"
         }
       ]
     },
     {
       "id": 12346,
       "name": "Part 2",
       "description": "No Description",
       "status": "Completed",
       "files": [
         {
           "id": 3,
           "extension": "txt",
           "path": "/SHA1/p2/file3.txt"
         },
         {
           "id": 4,
           "extension": "obj",
           "path": "/SHA1/p2/file4.obj"
```

```
49          }
50        ]
51      },
52      {
53        "id": 12347,
54        "name": "Part 3",
55        "description": "No Description",
56        "status": "Completed",
57        "files": [
58          {
59            "id": 5,
60            "extension": "txt",
61            "path": "/SHA1/p3/file3.txt"
62          },
63          {
64            "id": 6,
65            "extension": "obj",
66            "path": "/SHA1/p3/file6.obj"
67          }
68        ]
69      },
70      {
71        "id": 12348,
72        "name": "Part 4",
73        "description": "No Description",
74        "status": "Completed",
75        "files": [
76          {
77            "id": 7,
78            "extension": "txt",
79            "path": "/SHA1/p4/file7.txt"
80          },
81          {
82            "id": 8,
83            "extension": "obj",
84            "path": "/SHA1/p4/file8.obj"
85          }
86        ]
87      }
88    ]
89  }, {
90    "id": 3,
91    "name": "Sub Assembly 2"
92  }, {
93    "id": 4,
94    "name": "Sub Assembly 3"
95  }, {
96    "id": 5,
97    "name": "Sub Assembly 4"} ]
```

3.6.3 *Transfer Methods*

So far, we have covered how the application has access to the list of available Work Packages (Section 3.6.1), and after the selection of a Work Package which related Work Packages are marked for transfer (Section 3.6.2) through a detailed list of parts that comprise those Work Packages.

As mentioned before, HTTP must specify the content's extension/type of the message in its header `Content-Type`. Initially HTTP was designed to send one file per one request, but as we can deduce, after seeing Listing 2, we require sending multiple files and these files can have different formats between them. In this section, we will discuss two different methods to transfer these Work Packages and their files to the devices.

3.6.3.1 *Condense Method*

The simplest method for sending multiple files in one HTTP response is to compress all the files into a single compressed file, such as *zip* or *tar*. This way, we end up merely sending one file, whose type can be defined in the header, and contains all the requested files. For added protection, the compressed file can be encrypted under the content encrypted key described in Section 3.3.3.

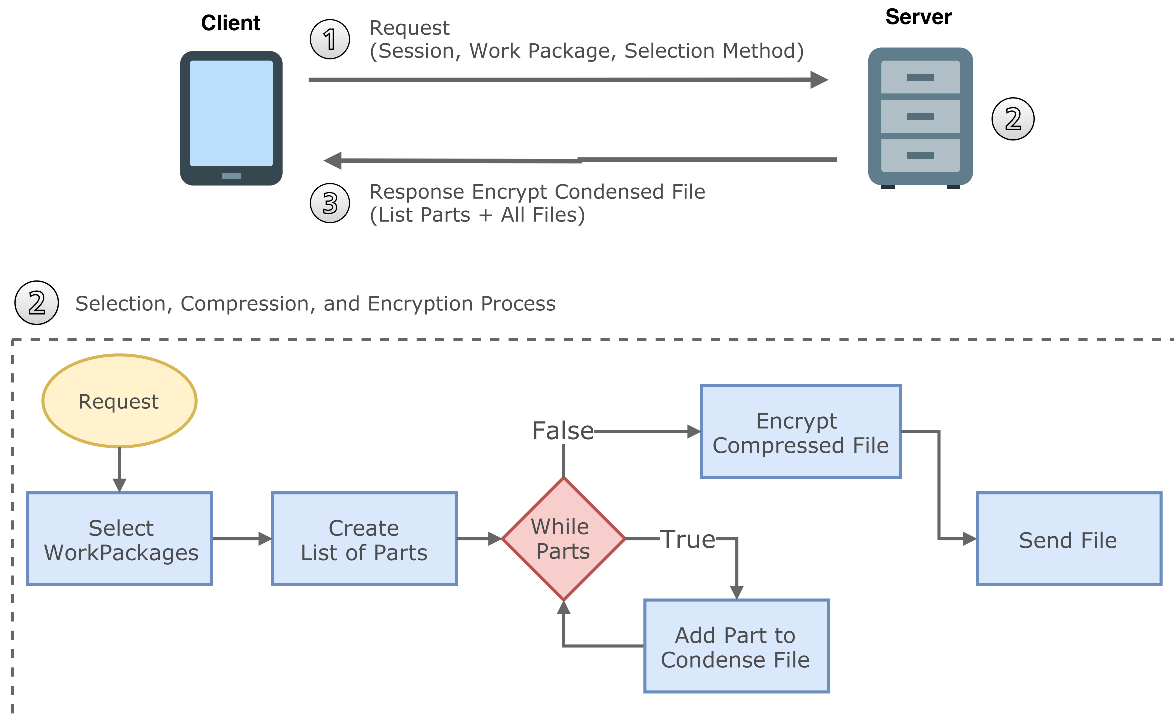


Figure 3.9: Condense Method Process.

The procedure can be seen in Figure 3.9. Corresponding Work Package files are read from disk and added to the compressed file. The internal structure of the compress file is the same as the location of the files in the filesystem. The process is repeated for all the files, until there are no more left. Afterwards, the compressed file is encrypted and sent to the application, where it is decrypted and decompressed.

The biggest advantage of this method is its simplicity. Any client application can deal with encrypted and compressed files, ensuring future compatibility. However, there are some drawbacks. The response containing the files can not be sent until the compressed file is created and encrypted. Combining with the time and computing resources it takes to decompress the file on the client side, this method can increase the response time and limited computational resources can be wasted here.

3.6.3.2 *Multiple-Requests Method*

Since HTTP is not fit for sending more than one file per request, a solution is requesting one file per request until all necessary files have been transfered. Depending on the number of files, this method can prove to be more effective than the previous, considering the time and resources spent dealing with compression are eliminated. Nonetheless, some difficulties/drawbacks can arise from making multiple requests. In Section 3.2 we explained that each connection to the server suffers from latency due to the connection establishment handshakes performed by TCP and SSL. If every request has to go through the same process, an overall increase in the response time can expect, Figure 3.10. A mitigation to these circumstances can be achieved by re-using the same underlying TCP connection for every request, thus having the handshake process being conducted only once for the overall files' transfer.

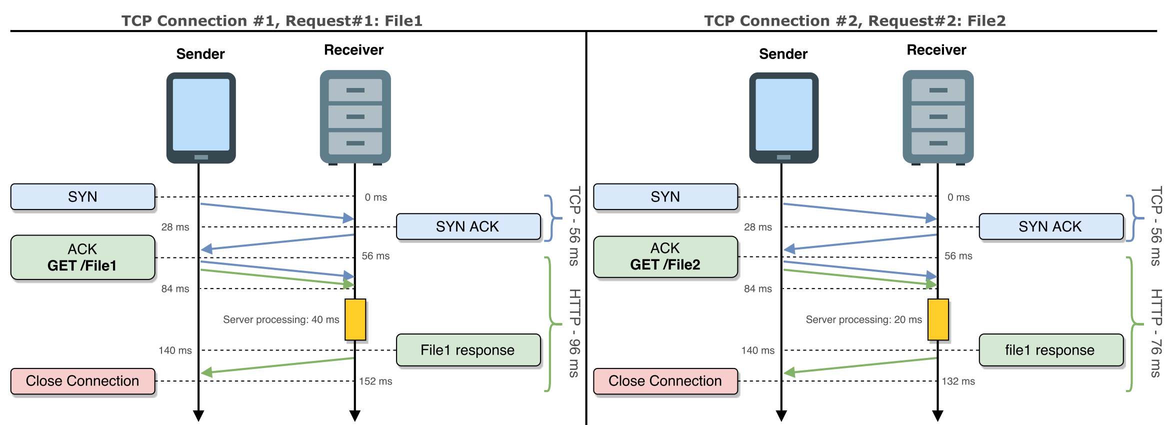


Figure 3.10: Two distinct TCP connections for each request.

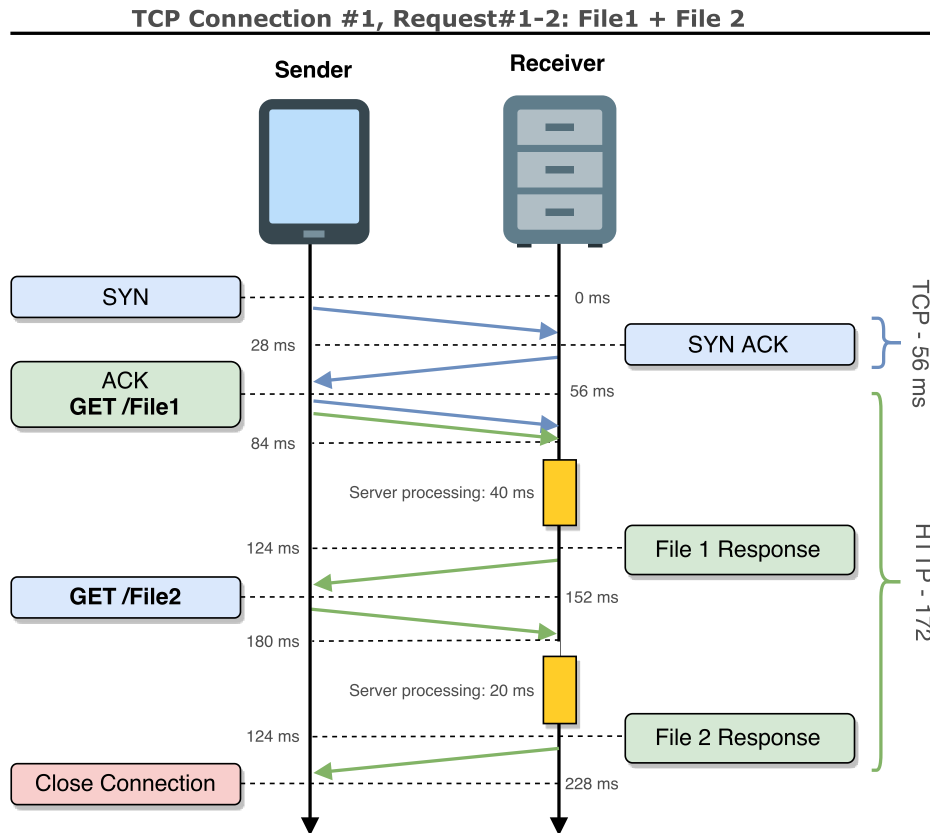


Figure 3.11: Multiple requests connection, HTTP reuse of underlying TCP connection.

3.6.3.3 *Method Comparison*

The efficiency of each method described previously is dependent on the amount of files being sent at a time. With the first method (Condense Method) files can be sent in one request, but consequently the client must first wait until the server has compressed all the files, before the client can receive them. Additionally, both server and client have to go through a compressing/decompressing process, which in case of a small number of files can result is wasted resources.

The second method (Multiple-Requests Method) on the other hand can send the files to the client the moment they are available, allowing the client to decrypt them while waiting for the next one. However, in comparison the decryption process must be performed for every file instead of just one which, depending on the number of files, can result in considerable amounts of time and computational resources being spent in this process. An advantage of this method is that, while the server is preparing to send the next file, the client can proceed to decrypt the file it just received, and continue to do so until receiving all the files. Moreover, if sufficient files have been transferred, the transfer process can

continue in the background, and the worker can start immediately with his task. Compared to the other method, the client has to wait until the server finishes processing the request, only then it will send a response containing all the files, and the worker can proceed with the assembly.

In Section 3.6.2, we introduced three methods for Work Packages' data selection. We concluded that section with an example of a JSON file — Parts List — which contains the information of the selected Work Packages, parts, and files. From the above descriptions, the Multiple-Requests Method seems to provide more flexibility and better performance when compared to the Condense Method. However, there is one big difference between the two. Since the latter compress all the selected data, the Parts List is not a requirement for the the actual transfer. The compression can take place immediately after the selection, without passing through the application. The Parts List JSON can be compressed inside the (compressed) file that is sent. On the other, for the former, the Part List is a requirement. The reason why is because with this method, the requests have to specify an exact file in the request, therefore the Parts List was to be sent prior to the transfer process, which can increase the overall transfer time.

In order to define which method is the faster, and therefore most suitable for our solution, it is required that both are implemented and tested with real-world files. Only after, we will have sufficient data to make an educated decision of which to support.

3.6.4 *Transfer Time Improvement*

For the sake of increasing the overall transfer time, we thought of two possible techniques. These techniques are described in the following sections.

3.6.4.1 *Location Identifier*

By examining the files' objects in Listing 2, we can see the path (in the file system) of each file, together with the rest of the file's information. But why? So far, an ID has been sent with a request and the ID of the files is also present, so why not continue with this scheme and simply send the ID? The reason is that to find the location of a file tied to an ID, a database lookup has to be performed to retrieve the location of the file, so it can be read and subsequently transfered. In Section 3.4.3, we debated the benefits of storing the files location in the database as oppose to storing a BLOB. Having the location of the file increases the size of Listing 2, but, in doing so, it opens the possibility to use the location of the file in the request, instead of the corresponding ID, skipping a database lookup entirely. A canonical path is always unique [92] so, for this case, it can also be used as an identifier. Saving one step in the transfer process will undoubtedly increase the speed at which the files are transfered.

Despite the fact that this technique can shave time off the overall transfer, it can only be applied in the Multiple-Requests Method. Because, in order to benefit from it, the Parts List must be sent, to the application, prior to the transfer, which is only necessary for the Multiple-Requests Method.

3.6.4.2 *File Caching*

In Section 3.5, we enumerated all the core components that form our sever, one of which is a cache component that can be used to cache files and requests, so that these can be sent faster to the client. The response time of the methods described in Section 3.6.3 can be improved if, instead of reading the files from the disk, they were read from memory, which can be up to 80 times faster.

On every request, specifically the ones for Work Pakcages/files, the server can search if the existing files, and if so read directly from it, instead of reading from the disk. In case it does not find it there, the server will proceed like before and read them from disk, saving them in memory afterwards. This logic is explained in greater detail in Figure 3.12.

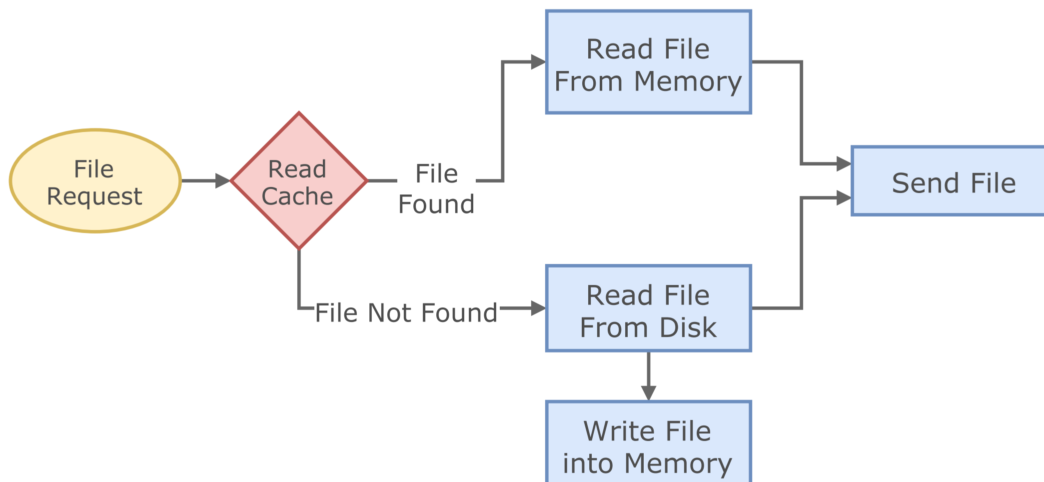


Figure 3.12: Cache Writing Policy.

However, it is not possible to store all files in memory, due the total size of the files and the memory's space limitations. There is a set of rules or algorithms that can be used to manage the information stored in cache. One of those is the **First-In-Last-Out**, Another algorithm is the **Least Recently Used**, For that, it requires keeping track of what file was requested and when it was used, which can be expensive if one wants to make sure the algorithm always discards the least recently used item. It also requires more data to be stored, an *age bit* that must be updated every time a file is used, which can add time to a procedure that is intended to save it. [93]

Another rule of thumb — **five-minute rule**

Multiple-Requests Method is, of the two methods, the one who benefits more from this improvement, especially when combined with the previous improvement (Section 3.6.4.1). Condense Method can still benefit since, as we mentioned, memory reads are substantially faster than disk reads.

3.6.5 *Instructions Update*

The major deciding factor in which of the commutation methods, described in Section 3.2.2, to choose was the ability to receive updates of assembly instructions. After examining the available communication methods and concluding the time for update requests updates, we decided that a simple periodic request for updates is an adequate design decision.

Depending on the Operating System (OS) or programming language, this process can have several names, but they are all part of a mechanism called time-based job scheduling, where jobs can be designed to run at a specific time or every hour/minute and so on. In our case, we want the application to periodically ask the server if the instructions currently being implemented have been updated.

As previously mentioned in Section 3.3.4, we introduced sessions into our system. Associated to each session we can have information related to the Worker and the Work Packages requested. As such, there are two possible design choices here: We can store all the Work Packages sent to the application, under that session together with the last time a Worker requested an update check; Or the application can provide this information in the request. Even if we decide to go for the latter, having the Work Packages stored in the server can help identify, at any given time, the Work Packages currently being assemble by a Worker. Nonetheless, if we recall the argument made previously about stateless, the system can be more flexible if the application provides all the information, thus reducing the implementation's complexity of server's session manager.

Figure 3.13 depicts the periodic request process just described.

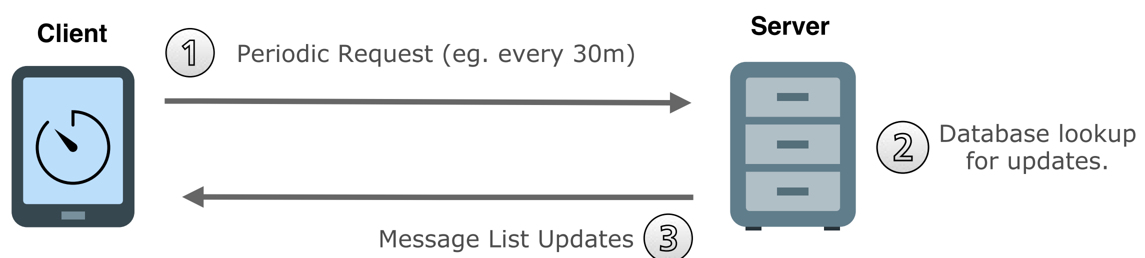


Figure 3.13: Periodic Instructions Update.

3.7 SERVICE MAPPING

In the context of our application, and in conformity with the project's requirements, the range of operations or services accessible by the client are described next. Some of which have already been covered.

AUTHENTICATION Authentication request for access to the server — Section 3.3.

ASSEMBLY INSTRUCTIONS Requests for specific Work Packages, and their corresponding parts and files. In Section 3.6, we investigated how exactly the transference of one or multiple files is conducted, and how precisely workers can have access to the full range of possible Work Packages.

UPDATES These requests will be sent periodically, as discussed in Section 3.6.5, to check if there are any new or updated assembly instructions to the ones currently being assembled.

STATUS Requests to change the status of both a part and/or a Work Package.

REPORT Report requests are sent when a worker reports a task as complete, ie., part(s) mounted, or report error(s) discovered during the execution of that task.

Each of these operations must be addressable *via* an Uniform Resource Identifier (URI) or its subset URL, so they can be accessible by the client. These operations correspond to our API and expose our business logic to the world. Table 3.6 contains the list of services, together with the URI, request method, headers, body, response to the request, and a brief description. In this table, request path parameters are marked with "{ }". Different resources or services are provided depending on the method or the parameter passed. For example, the path parameters for the request `/wp/{id}/{selection}`, which correspond to the Condense Method, correspond to the ID of the Work Package and the selection method requested. Afterwards, in response, a compressed file containing the files and the JSON List Parts is sent.

In Chapter 5 we explain how each of these services and every design decision discussed in this chapter was implemented.

Table 3.6: Service Mapping.

Resource URI	Method	Header	Body	Response	Description
/worker/login	POST	-	Credentials	Session Key	Credentials are sent in the body of the request. If authenticated, a session key is returned.
/worker/logout	POST	Session	-	Message	Session is removed from the server.
/wp/listWP	GET	Session	-	List Work Package (JSON)	List of Work Packages avaiable
/wp/{id}/ {selection_method}	GET	Session	-	Compressed File	Condense Method File. The content depends on the selection method chosen. The List Parts JSON file is also included in the file.
/wp/{id}/listParts/ {selection}	GET	Session	-	List Parts (JSON)	List of Parts. Necessary for the Multi-Request Method. Sent inside the compressed file in Condense Method.
/wp/file/{path}	GET	Session	-	File	Multi-Request Method. By using the file path as the identifier, we can skip a database lookup.
/wp/updates/	GET	Session	Work Packages & Last Update Time	Work Packages Updated	List of Work Packages that have been updated. If no information is sent in the body,the data saved (in the server) under the session is used.
/wp/{id}/status/	PUT	Session	Status	Message	Change Work Package status to the value sent in the body
/part/{id}/status	PUT	Session	Status	Message	Change Part status to the value sent in the body
/part/{id}/mounted	PUT	Session	-	Message	Check Part as mounted by the owner of the session.
/part/{id}/report	PUT	Session	Report Message	Message	Report problems or other remarks about a part.

4 FRAMEWORK

> *"You just need a framework and a dream."*
>
> Michael Dell

Before moving on to the implementation of the architectural plans described in Chapter 3, the present chapter will briefly explain some of the technologies used in the implementation chapter, including database, and both server and client side technologies.

4.1 DATABASE AND DATASTORE

In this section, we describe the relational database essential for the storage and organization of assembly instructions and other pertinent information, and the cache memory database.

4.1.1 *MySQL*

MySQL is a free open source Relational Database Management System (RDMS). It works as a server and can be managed either via command line or via client applications. As a relational database, the data is organized into different tables. In each table, the information is stored as entries (rows in a table). Each entry can be related to many entries in the same or in other tables. The relationship between entries is made through a foreign key, which is an unique identifier of an entry in a table. Queries in a MySQL database are done through Structured Query Language (SQL), a standardized language used to query relational databases.

4.1.2 *Redis*

Redis is a fast stand alone open source (BSD licensed), non-relational in-memory data structure store, used as database, cache and message broker. [95] It supports several data structures like strings, lists, and hashes. It is capable of in-memory, persistence storage on disk, and replication distribution. It is the ideal place for storing sessions since data can be restarted and redeployed in case of failure. [96] All these reasons are what leads up to use Redis in our solution.

4.2 SERVER FRAMEWORK

The following sections introduce the technologies and tools that were used during the implementation of the server side.

4.2.1 *Node.js*

The Node Foundation [97], a collaborative project at the Linux Foundation [98], is responsible for the host, adoption, and development of Node.js. Node.js is an open-source, asynchronous event driven framework, cross-platform runtime environment for developing scalable network applications [99]. Applications that are written in JavaScript, one of the essential technologies of the WWW. Having a unified language for both client and server development, which can also be used for building, testing, and templating, reduces the overhead of dealing with multiple systems and languages (libraries can be shared across the stack) and the number of developers required for a project, as well as the overall complexity of the process when exchanging messages. Another advantage is Node's runtime ability to be executed on a number of OS's, such as Mac OS, Microsoft Windows, and Linux.

Node's thesis/proposal/opinion is that Input/Output (I/O) must be done differently. For example, most of the server programming languages deal with I/O operations the following away: a database query is performed; the result is returned; the program uses the result. In many cases, while performing the database query, the server hangs waiting for the result response, and when dealing with I/O latency, either by disk, memory, or network latency, the server cannot just wait for it to respond. For these reasons, many programming languages employ multi-thread techniques to improve performance and concurrency, although these applications are not easy to write, debug, and manage [100]

Node operates on a single thread (although multi-thread is possible), so if it were to hang, the entire server would hang awaiting for the result, thus it should be able to multitask. The main reason for using Node.js in the implementation of our solution is Node's event-driven architecture and non-blocking I/O API, which is designed for optimization, throughput, and scalability. If Node were to perform the database query described previously, it would issue the query and immediately process something else. The moment the query result is returned, it triggers an asynchronous callback function responsible for handling those results, allowing it to handle many connections concurrently. [101] [102]

On the matter of performance, Jason Hoffman, Chief Technology Officer (CTO) of Joyent, the company where Node.js was created, stated that a single core with less than 1GB of RAM is fully capable of handling 10 Gbps of traffic and one million connected end points. [103] This makes Node well suited for the foundation/basis of our solution.

4.2.2 *Node Package Manager*

Node Package Manager (npm) is a public code registry open to all developers and where code can be shared and reused. These bits of reusable code are called **packages** or **modules**. An application is formed by multiple packages, and the freedom to re-use them allows developers to draw expertises from the outside and bringing on packages that are focused on particular problem areas. Also, code can be freely used in other projects by importing the necessary packages.

Npm makes it easy for packages to stay updated to the latest versions and to set up developing environments across teams. The following paragraphs describe the major packages used in our solution.

EXPRESS.JS Express.js [104] is a minimal and flexible web framework, and the base for the majority of Node.js applications. It is based on the core Node.js HTTP module and Connect packages, which are typically referred to as *middleware*. Express.js is the cornerstone of our server, where all components and code can be elegantly structured under a Model-View-Controller (MVC) organization. Express popularity is due to its fast learning curve, easy to implement routing API, and request handling (Controller). [105] [106] Other advantages of this framework are based on simplifying difficult tasks like parsing HTTP requests (URL, parameters, and bodies) and cookies, managing sessions, handling errors, and proper response headers based on data types.

SEQUELIZE Sequelize is a promise-based easy to use multi SQL dialect Object-Relational Mapper (ORM) for Node.js [107]. It currently supports a variety of RDMSs like PostgreSQL, MySQL, MariaDB, SQLite and MSSQL. An ORM makes data access more abstract and portable, since classes are defined through an ORM and not subjected to vendor-specific SQL. Thus, transitioning from one database to another does not require all transactions and queries to be re-written.
As mentioned before, Node does not come equipped with database clients. Sequelize, on the other hand, comes bundled with database specific client protocols for the databases it supports.

REDIS-CLIENT Since Redis is not a relational database, Sequelize can not be used together with our Redis database. For this reason, we also require a Node.js Redis client that can support all Redis commands, and focuses on delivering the high performance provided by Redis.

HAT Small (1.6 KB) package used to generate random 128-bit long session keys and avoid collisions in this procedure. [108]

BLUEBIRD “Fully featured promise library with focus on innovative features and performance”. [109] Bluebird can "promisify" an existing promise-unaware API or function into a promise-returning one.
Promises are a software abstraction that makes it easier to work with asynchronous operations. A promise represents the future result of an asynchronous operation. They make it easier to *throw* and *catch* errors or exceptions and have proven to make code easier to read and maintain. Nevertheless, the more important aspect for any javascript developer is, perhaps, the ability to eliminate the fearful callback hell [110]. The use of Promises can be seen in 3 however, in case the reader is interested in learning more about Promises, we recommend reading [111] [112] [113].

ARCHIVER In order to test the method described in Section 3.6.3.1, we require a package capable of creating compressed archives. Although there are many archive or data compression generators packages for Node.js, Archiver was chosen for being a “streaming interface for archive generation” [114] capable of creating archives in formats like zip and tar.

4.3 CLIENT FRAMEWORK

The following sections regard the technologies and tools used during the client side implementation.

4.3.1 *Android Operating System*

The current application is built over the Android OS, currently developed by Google. Android is a rich mobile application framework for building mobile applications in a Java language environment. [115] It has an adaptive device configuration system, which supports the creation of different layouts for different screen sizes, such as phones and tablets.
For this thesis, the two fundamental Android components are:

ACTIVITIES This component represents a screen within the user interface. Applications usually consist of one or more activities. Activities can work together and pass information during the transition from one to another, however they are lossy bound to each other, meaning each one is independent. [115]. During transition, the previous activity can be terminated or stopped. A stack keeps track of which activities have been stopped and pushed to the back in a "Last-in, First-Out" fashion. The state of these transitions is tied to the activity lifecycle and can be managed by implementing callback methods. [116] The activities’ lifecycle and the callback methods can be seen in Figure 4.1.

SERVICE This component runs a process in the background of the application. It is normally used to “perform long-running operations or perform work for remote pro-

cesses” [115]. In the context of this thesis, services are used to periodically send assembly instructions update requests. A service must be started by another component, such as an Activity, however it is not bound to the same lifecycle of the component that started it.

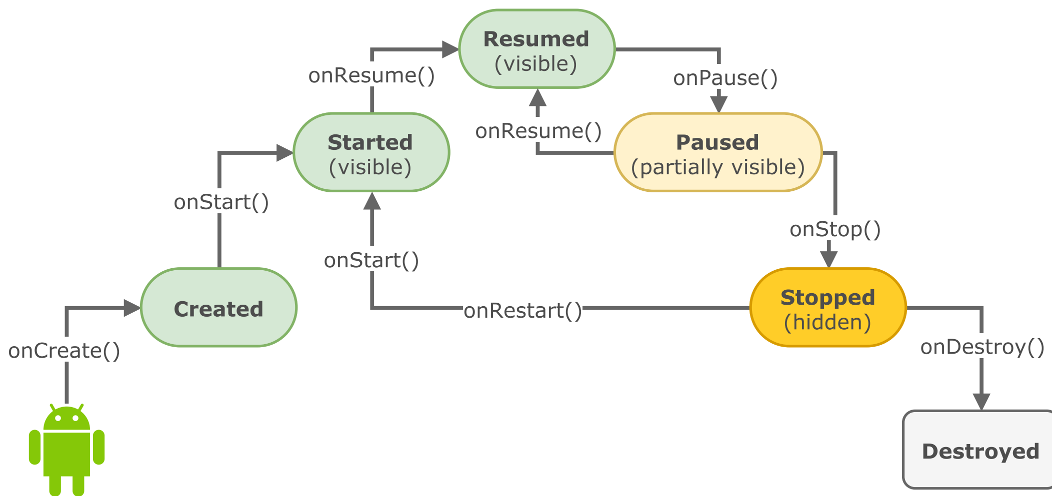


Figure 4.1: Android Activities Life Cycle, adapted from [117].

Components have to be initiated in the main thread, sometimes called "UI Thread". Because of this single thread model, In order to perform asynchronous operations, for instance request to the server, worker threads must be created to avoid blocking the UI Thread. This is accomplished *via* an **AsyncTask**. When worker thread functions are finished, the results can be published on the the UI Thread without any handling of threads being required.

4.3.2 *Android HTTP Client*

There are an assortment of HTTP client libraries available for Android. Android itself comes equipped with two HTTP clients: HttpURLConnection and Apache HTTP Client. However, we found these libraries too complex to be used in our solution. Developed by Square, **Retrofit** is a “type-safe HTTP client for Android and Java” [118]. Retrofit provides annotations for request declaration and class generation. Synchronous or asynchronous HTTP requests are easily defined and it has support for SSL, URL parameter replacement and request body conversion. Underneath, it uses **OkHTTP**, another library developed by Square, which features silent recover from common connection problems, connection pooling, response caching, and the ability to fully manipulate an established connection.

5 IMPLEMENTATION

"A good idea is about 10 percent and implementation and hard work, and luck is 90 percent."

Guy Kawasaki

In this chapter, we will demonstrate how the architectural plan in Chapter 3 was implemented, using the technologies described in Chapter 4. We will explain our model, how data is specifically exchange, and how its security is assured. The following sections are also completed with contextual examples and descriptions. Not every portion of the implementation will be covered, only the core parts of the database, the establishment of the system, and the transfer methods. The renaming parts of the implementation can be seen in the CD that accompanies this thesis.

5.1 DATABASE IMPLEMENTATION

This section will focus on the implementation of our model as a database and according to the ER diagram Figure 3.3. We will also address the structure of the Work Packages' dependencies table, and how data is validated based on the structure deployed.

5.1.1 *Database Structure*

Previously, we designed a crude generalization of the information pertinent to the construction of one or multiple ships — Figure 2.12. After careful analysis of the model's relational representation, we arrived at the database design represented in Figure 5.1.

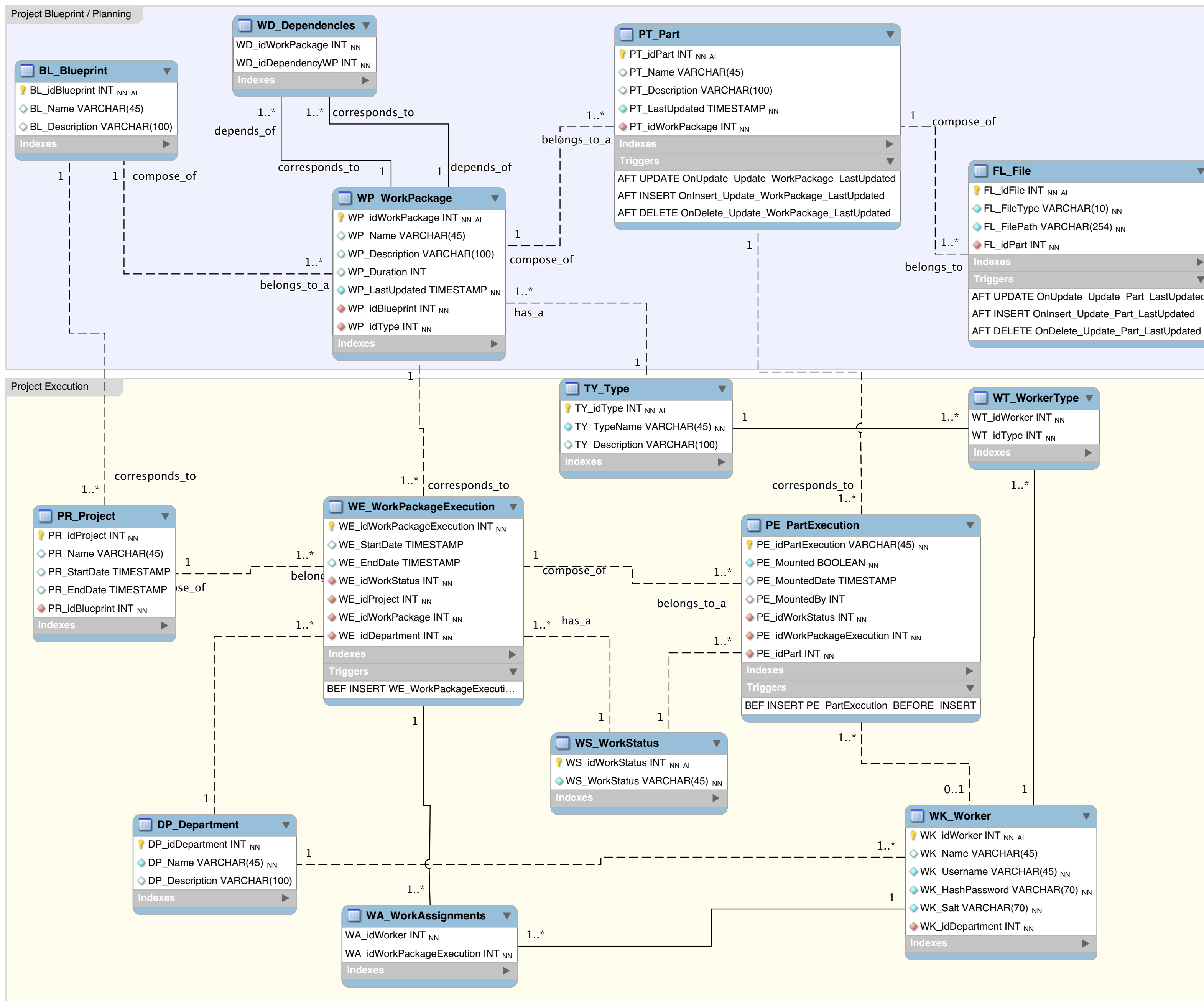


Figure 5.1: MySQL database

At a first glance, we can notice that the representation depicted in Figure 3.3 does not exactly match the representation seen in Figure 5.1. The database was separated into two groups: **Blueprint** and **Execution**. This division is necessary so we can prevent data redundancy when two or more identical ships are to be assembled. In the upper part, Blueprint, tables represent how a Work Package structure of a ship is organized and related to the planning for a specific ship's design. Bellow, in the Execution group, are the tables related to the execution of that plan. For this reason, time related attributes are only present in the tables of this group, as well as the status, assignments, and department where a specific Work Package, now a execution of a Work Package, should take place. Following this logic, we can identify that the same relationships presented in Figure 3.3 are also present in the database, in addition to relationships between brother tables, like *WP_WorkPackage* and *WE_WorkPackageExecution*. Having this division, will allow for more control over the progress and distribution of Work Packages.

Besides the division mentioned above, another aspect obvious in Figure 5.1, is the declaration of the attributes types, and if an attribute is required, meaning must be filled (`not null`).

A best practice when dealing with timestamps is to always store data as Coordinated Universal Time (UTC). Many database do date/time and interval arithmetics in UTC time, and only after, convert it back to the database time zone. [119] UTC time does not observe daylight savings time, thus being considerered interchangeable. Time zones can be handled in the application, avoiding time zone and daylight savings conflicts in the database.

5.1.2 *Work Package Dependencies*

Regarding the hierarchical representation of the Work Packages, there are certain patterns for the model of hierarchical data that must be taken into account. The pattern representation most commonly seen is the inclusion of a parent attribute. However, this does not fully represent the existing structure because, as we have seen by now, the dependency structure of a Work Packages is more similar to a graph than a tree structure. An ideal pattern should be one that can have an easy implementation and maintenance during changes — Insert, Delete, and Update — and can preserve the referential integrity [11] of the Work Packages records. [121]

Bill Karwin explains, in his book [122], four different patterns for hierarchical data designs: Adjacency Lists; Path Enumeration; Nested Sets; and Closure Table. The description and comparison of these patterns can be seen in Table 5.1.

The analysis of the hierarchical data designs, depicted in Table 5.1, quickly reveals which of the four patterns is the best. **Closure Tables** have easy implementations features, maintain referential integrity, and have quick and easy methods for querying ancestors and

11 Referential integrity means that a database must not contain any unmatched foreign key values. [120] Every foreign key has a corresponding primary key.

Table 5.1: Hierarchical data designs comparison. Adapted from [122].

Design	Tables	Query Child	Query Tree	Insert	Delete	Ref. Integ.
Adjacency List	1	Easy	Hard	Easy	Easy	Yes
Path Enumeration	1	Easy	Easy	Easy	Easy	No
Nested Sets	1	Hard	Easy	Hard	Hard	No
Closure Table	2	Easy	Easy	Easy	Easy	Yes

Table 5.2: Representation, through closure pattern, of the records corresponding to Work Package S2.

Work Package	Work Package Dependency
S2	S1
S2	U1
S2	U2
S2	U3
S2	U4
S2	U5
S2	SA1
S2	SA2
S2	SA3
S2	SA4

descendants. In Figure 3.3, we can confirm the implementation of this pattern, by the creation of the two necessary tables. Table *Work Package* has a many-to-many (`N - N`) relationship with itself generating, as a result, the table *Dependencies*. These tables belong to the Planning portion of our database.

In a closure table pattern, every (dependency) path is stored from each Work Packages to each of its dependencies, meaning that not only direct dependencies are stored but also the dependencies of those dependencies and so on. Taking Work Package S2, from our example depicted in Figure 3.8, the records stored will coincide with those present in Table 5.2. Data represented with this pattern results in an $O(n^2)$ number of rows. Still, size is a minor compromise compared to the performance obtained in the selection process.

5.1.3 *Data Validation*

In Figure 5.1 we can observe that there are *triggers* associated with some tables. *Triggers* are database objects that activate when certain events occurs for that table, particularly when a (SQL) statement inserts, updates, or deletes rows/records in the associated table.

The *triggers* present in the database are divided into two groups:

TIMESTAMPS UPDATES In tables *WP_WorkPackage*, *PT_Part*, and *FL_File* triggers ensure that fields related to dates *(LastUpdated)* are always up-to-date. When changes occur to a certain record, all the associated records, through the use of *Foreign Keys* [12], suffer alterations. However, these alterations only happen in reverse order, i.e., from the many part of the relationship to the one. For example, if a record in table *FL_File* is inserted, updated, or deleted, the associated records in the other two tables, *WP_WorkPackage* and *PT_Part*, will have their data time updated. However, when a change occurs to a record in *WP_WorkPackage* the same does not happen the other way around. The reason is because it is unnecessary to update these tables, considering the selection factor are the work packages, as we described in Section 3.6.2.

ASSOCIATION VALIDATION This group of *triggers* are present in the *Planning* section of the database, in tables *WE_WorkPackageExecution* and *PE_PartExecution*. Relationally speaking, any record in *WE_WorkPackageExecution* can have a reference to another record in *WP_WorkPackage*. However, there are cases where such reference relationships cannot be allowed. Figure 5.2 portraits a scenario of that nature. A Work Package that belongs to a specific Ship's Blueprint cannot be referenced by a Work Package *Execution* that is affiliated with any other than the same Project's blueprint.

Listing 3 corresponds to the SQL code that enforces that validation. The trigger present in Table *PE_PartExecution* shares a similar code.

12 Foreign key is a (collection of) field in a table that (uniquely) identifies a another field (Primary Key) of another table.

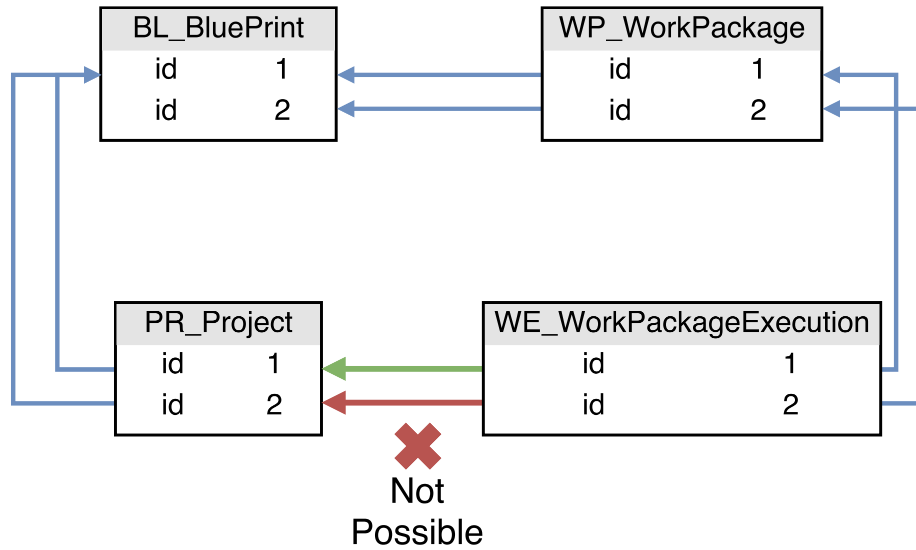


Figure 5.2: Trigger Association Validation

Listing 3: Records Association Trigger

```
CREATE DEFINER = CURRENT_USER TRIGGER `thesisdb`.`WE_WorkPackageExecution_BEFORE_INSERT` BEFORE
    INSERT ON `WE_WorkPackageExecution` FOR EACH ROW

BEGIN
  DECLARE bl INT;
  DECLARE pr INT;

  SET bl = ( SELECT BL_idBlueprint
    FROM WP_WorkPackage JOIN BL_Blueprint ON WP_idBluePrint = BL_idBlueprint
    WHERE WP_idWorkPackage = NEW.WE_idWorkPackage );

  SET pr = ( SELECT DISTINCT PR_idBlueprint
    FROM WE_WorkPackageExecution JOIN PR_Project ON PR_idProject = WE_idProject
    WHERE WE_idProject = NEW.WE_idProject);

  if (bl != pr) then
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Validator WorkPackageExecution FAILED';
  end if;
END
```

5.2 system configuration

This section will describe, in greater detail, how many of the architectural designs present in 3 were implemented. This and further sections will focus solely on the system's configuration, methods' declaration, and how the more crucial methods were implemented. Examples will be provided, however these may be incomplete for brevity. As mentioned before, the CD that accompanies this thesis contains all the project's code, and the complete implementation can be found there.

5.2.1 *System Modular Design*

The server side of our system follows a modular design that separates the functionalities of our program into independent and interchangeable modules. By employing this design, we can have a clear separation of the different components; algorithms are also more direct and understandable; and updates or changes can easily be introduced without involving major changes to the whole program.

In Figure 5.3, we can see an overview of the modular system implemented to the server.

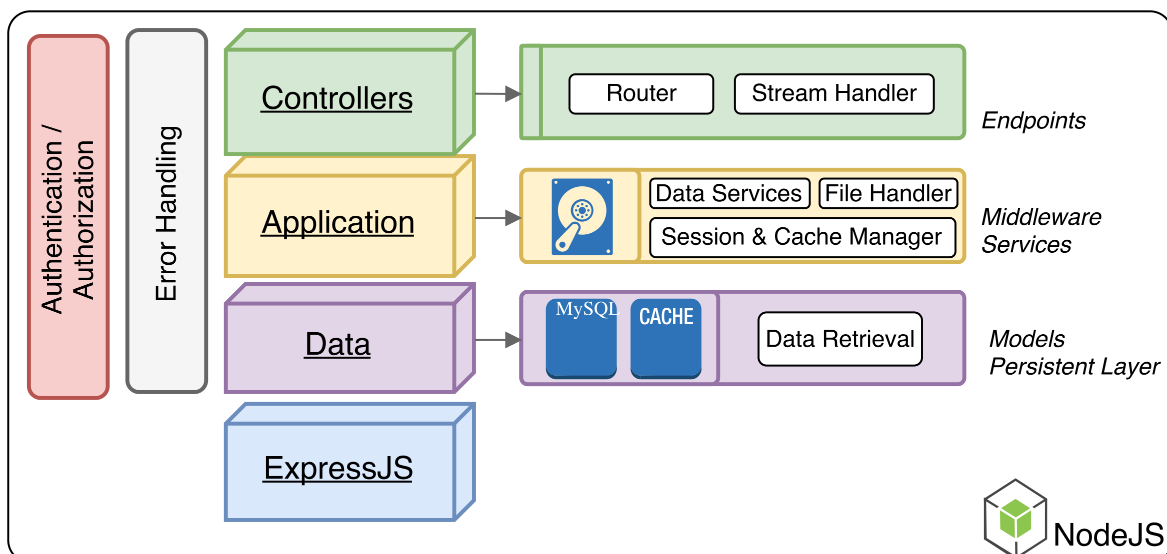


Figure 5.3: System abstract overview. Three different layers, under a modular design structure. All layers are connect to error handling and access control mechanisms

This multi-layer modular design is partially based on the Law of Demeter [123], or principle of least knowledge, where methods should avoid invoking objects that are returned by another method. [124] Moreover, each layer has only the minimal knowledge about methods in other layers, knowing only enough to accomplish its function.

Three distinct layers are represented In Figure 5.3:

CONTROLLERS LAYER Its main function is to translate tasks/requests and results to something the user can understand. Using Express.js, on top of Node.js, facilitates the connection of all the *middleware*, and allows us to make use of its incredible routing abilities. This layer acts as the facade of our server, i.e., exposes the services, and delegates the logic to the business layer. The services available were defined in Table 3.6.

In Section we will how to declare the controllers, and in Section Requests that

Besides the controllers, the stream handler methods are also presented here, we will examine those methods in Section

APPLICATION LAYER Sometimes called business, logical, or middle layer, its the layer responsible for controlling the application and making logical decisions based on the business logic implemented. Moreover, it moves and processes data between the two surrounding layers. In summary, this layer implements the core functionality of the system, and encapsulates the relevant business logic that dictates how requests are processed. [125]

DATA LAYER This layer provides access to data hosted within the boundaries of the system. In our case, this data is present in the database and cache server. The Data layer includes the data persistence mechanism that encapsulates and exposes these data. Using an ORM allows us to create methods for managing data without exposing or creating dependencies on the data storage mechanisms of a specific system/server. The multi-layer architecture ensures that if there are ever any changes or updates to the methods in this layer, the above layers do not require any modifications, as long as the returned objects stay the same. [125]

5.2.2 *Server Configuration*

This section focuses on the server configuration, how components were connected, and the different techniques implemented to improve the server performance.

5.2.2.1 *Package Declaration*

All Node.js projects and packages have, at the root, a file named *package.json*. This file contains metadata about the project, and is used by npm to identify the project, as well as its dependencies. Thanks to this file all the project's packages dependencies can be loaded safely no matter the environment in which the project is deployed. Listing 4 depicts the *package.json* file of our project.

Listing 4: Package.json

```
1 {
2   "name": "Server_Express_App",
3   "version": "0.0.1",
4   "private": true,
5   "scripts": {
6     "start": "node ./bin/www"
7   },
8   "dependencies": {
9     "archiver": "^0.15.1",
10    "bluebird": "^2.10.2",
11    "body-parser": "~1.13.2",
12    "crypto": "0.0.3",
13    "debug": "~2.2.0",
14    "express": "~4.13.1",
15    "hat": "0.0.3",
16    "redis": "^2.1.0",
17    "sequelize": "^3.10.0",
18  }
19 }
```

When executing the project, a command specifying the starting script must be provided as argument. In order to facilitate this task, inside *package.json*, we can define the scripts properties for different actions. To start the project, we defined in Line 6, the command that initiates the project.

Lines 8 to 18 represent the project's dependencies, which were described in Chapter 4. Even though, we use Express.js (Line 14) and Node.js in our project, not all of their packages are required. The packages in Lines 11, 12, and 13 belong to packages embed into Node and Express. Their purpose is self explanatory. They are declared so that only these, and not all packages inside Node.js and Express.js, which are not necessary, are imported. Otherwise, the size of the project would increase exponentially if all the packages inside Node.js and Express.js were defined.

The name and version of the packages form an identifier which, inside npm, are completely unique. Having the versions of the packages prevents deprecated methods from occurring, and ensures that everything will work as designed.

5.2.2.2 *Redis and MySQL Connection*

Node.js has a simple and practical module loading system. Modules encapsulate code into a single file, that can then be used by other modules or methods. Modules help in keeping the code separated and organized. In Listings 5 and 6 we can see how the connection with the MySQL and Redis server is created. In each Listing, we can see the clients created with a set of defined options. At the end of each, these clients are exported, *via* `module.exports = <variable>`. That way, they can be easily used by methods in other files when required.

These variables can be used in other files *via* `module.exports = <package or file>`. We can see it in use, in the beginning of each Listing, when invoking the project's module.

Listing 5: Redis Client.

```
1  var redis = require('redis')
2
3  var client = {
4    sessionDB: redis.createClient('127.0.0.1', '6379').select(1),
5    filesDB: redis.createClient('127.0.0.1', '6379').select(2, {
6        return_buffers = true
7    }),
8  };
9
10 module.exports = client;
```

Listing 6: Sequelize Client for MySQL

```
1  var Sequelize = require('sequelize');
2
3  var database = 'thesisdb';
4  var username = 'root';
5  var password = 'root';
6
7  var sequelize = new Sequelize(database, username, password, {
8    host: 'localhost',
9    dialect: 'mysql',
10   pool: {
11     max: 20,
12     min: 0,
13     idle: 10000
14   }
15 });
16 module.exports = sequelize;
```

The code present in Listings 5 and 6 is self explanatory. In Listing 6 we declare the Sequelize ORM (Section 4.2.2), with the location of the server, and the type of RDMS, MySQL in our case. In this Listing, there is also one small technique to increase the performance of the communications between Node and MySQL. By default, only one connection exists between Node and the databases MySQL and Redis. However, by using a **Connection Pool**, a cache of connections can be used by the application to access the database server. Using a connection pool can increase the performance of our system, since the time required to establish a connection is eliminated. [126] In Listing 6, Lines 10 through 14, we can see the maximum number of connections (20) created between Node and MySQL. These connections are destroyed after an idle period of 10000 milliseconds or 10 seconds, to free resources. This way, whenever Node wants to access the MySQL server, a connection is

already ready to be used, if a new request to the database is issued, another connection is already established and ready.

One client per application is an efficient pattern, however there are occasions in which having more than one connection is beneficial. Since both Node.js and Redis are effectively single thread there is however, no beneficial gain in using a connection pooling when connecting to the Redis database.

If we perform a close analysis of Listing 5, we notice two different redis clients. Redis' servers can have multiple databases in one instance. To separate the session management and the file's cache bind each client to a different redis databases in the same server (Lines 3 through 8).

5.2.2.3 *Sequelize Models*

We opted to use an ORM, Sequelize, so that data access methods can be defined in an abstract manner, and be portable to other RDMS dialects if the need occurred. An example of the Blueprint's table model, can be seen in Listing 7. The model corresponds identically with the table depicted in Figure 5.1. We can identify all the attributes of the table Blueprint, their type, the primary key, and if they allow nulls or not. At end, the relationships this table has with others are defined, in affinity whit the type of relationship.

Listing 7: Sequelize Blueprint Model

```
1  /* Code omitted for brevity */
2
3  var BluePrint = sequelize.define('BL_Blueprint', {
4    BL_idBlueprint: {
5    type: Sequelize.INTEGER(11),
6    allowNull: false,
7    primaryKey: true
8    },
9    BL_Name: {
10     type: Sequelize.STRING,
11       allowNull: true,
12       defaultValue: 'No Blueprint Name'
13   },
14   BL_Description: {
15     type: Sequelize.STRING,
16     allowNull: true,
17     defaultValue: 'No Blueprint Description'
18   }
19   }, {
20       tableName: 'BL_Blueprint', // Table name
21   });
22
23 /* Blueprint Relationships */
24 BluePrint.hasMany(workPackage, {foreignKey: 'WP_idBlueprint'});
25 BluePrint.hasMany(project, {foreignKey: 'PR_idProject'});
```

```
26
27 module.exports = BluePrint;
```

Listing 8 shows the abstract nature of a method using Sequelize. The example depicted shows how a blueprint can be retrieve by its ID. In case, there is ever a change to a new dialect, this and other methods will not require changes. The object returned is an object model that has the same structure of the one defined previously in Listing 7.

Listing 8: Sequelize FindbyId BluePrint

```
1 var getBlueprintById = function (blueprintId) {
2   return blueprint.findById(blueprintId).then(function (result) {
3       return result;
4   }).catch(function (error) {
5       throw error // <- throws the error
6   });
7 };
```

5.2.2.4 *Application Configuration*

As we explained in Section 5.2.2.1, we defined, in the *package.json* file, a script to initiate our program, `node ./bin/www`. In this file, which can se seen in Listing 10, we initiated the server with the application's configuration shown in Listing 9. In Listing 9, we instantiate an express application (Line 3), establish the request handling packages (Lines 5 and 6), and the API routes (Lines 9 trough 11) defined in Table 3.6, with which one pointing to a different file related with that URI.

Listing 9: Express App configuration. Routes definition

```
1  /* Code omitted for brevity */
2
3  var app = express();
4
5  app.use(bodyParser.json());
6  app.use(bodyParser.urlencoded({extended: false}));
7
8  // Routes
9  app.use('/worker',    require('./routes/worker'));
10 app.use('/wp',        require('./routes/workpackage'));
11 app.use('/part',      require('./routes/part'));
```

Listing 10: Node.js Server

```
1 /* Code omitted for brevity */
2
3 var app = require('../app');
4 var https = require('https');
```

```
5  var fs = require('fs');
6
7  // Get port from environment and store in Express
8  app.set('port', normalizePort(process.env.PORT  '3000'));
9
10 // Create HTTPS server with credentials
11 var key = fs.readFileSync(__dirname + '/key.pem', 'utf-8');
12 var certificate = fs.readFileSync(__dirname + '/key-cert.pem', 'utf-8');
13 var options = {
14     key: key,
15     cert: certificate
16 };
17
18 var server = https.createServer(options, app);
19
20 // Listen on provided port, on all network interfaces.
21 server.listen(port);
22 server.on('error', onError);
23 server.on('listening', onListening);
```

As we discussed in Section 3.3.2, we will be using SSL to secure the communication between the server and the client. To avoid the costs associated with a SSL certificate, we will be generating a self-signed SSL certificate meant only for testing. Self-signed certificate means we certify this certificate instead of a CA. To avoid trust warnings, the certificate should be added to the application, or to its Trust Certificate. A `pem` file, which includes an entire certificate chain including public key, private key, and root certificates [127], is read and used in the server object options (Listing 10, Lines 11 through 16).

5.2.2.5 *Request Routes and Authentication*

In the previous section, we understood how the mapping of the routes can be easily defined using Express.js. Each route defined in the application file (Listing 9) refers to a different file, where the reaming path of the URI is handled. An example can be seen in Listings 11. The method is a simple test request, used only to test the authentication of the requests sent to the server, and is not present in the submitted version. Nevertheless, it is a simple example to explain how requests are handled.

Listing 11: Route GET request

```
1 var express = require('express');
2 var router = express.Router();
3
4 var validator = require('../utils/requestValidator');
5
6 router.get('/worker/test', validator.validateSessionReq, function (req, res) {
7   res.json("You are a real Worker!")
8 });
```

In Line 6, we can see Express' router function for a HTTP `GET` method request bound to the `/worker/test` URI. Before the request is processed, it must pass validation, to verify if the session exists. Only after this, does the server process the request, sending a JSON message in response.

5.2.2.6 *Messages Format*

There are several types of messages that can be sent to the client. For the most part, we follow a particular message format design, that can transmit as much information as possible. This is particularly useful when dealing with error messages. Listing 12 shows an example of an error message.

Listing 12: Package.json

```
1 {
2   "sucess": false,
3   "uri": "http://192.168.10.72/wp/updates/",
4   "payload": {
5     "status": 404,
6     "code": 3810,
7     "message": "Error Found",
8     "developer_message": "An error occur while performing a search in the database for a work
        package update with id = 31"
9   }
10 }
```

The payload can change depending on the success of the request. In case a request is successful, application-specific data should be inserted in the payload object. The data depends on the request made at the time. For error messages, the format should be similar to the one described before. The payload in Listing 12 shows the HTTP status code, an internal company code for that specific error message, a message that can be passed to the application, and a special message meant to help developers fix the errors that can occur when receiving this message.

5.2.2.7 *Redis Database Structures*

In Section 5.2.2.2, we described the creation of the connections to our databases, MySQL and Redis. We mentioned also, that two separate redis databases: one to store and manage the sessions; and another solely to the cache of CAD files, are used. Redis is not a plain key-value store, but actually a data structure server. In this section, we will explain the structure of both redis databases used in this system.

Traditional key-value stores, are limited to strings, however redis can deal with more complex data structures. From the database that deals with sessions we are going to use

For every session, we have the correspondent map of fields:

WORKERID The ID of the worker for which the session key has generated.

LISTWORKPACKAGES The list of work packages requested by this session.

CREATEDAT The session key generation time.

LASTUPDATE The last time a request, with this session, requested an update of the work packages. We mentioned before that in case the request does not provide the necessary information, that the values stored in the value under this field and the Work Packages in *ListWorkPackages* are used to process the request.

Listing 13 shows the login service function, where the logic defined in Figure 3.4 is implemented, and a hash is created and inserted into the database. This Listing also shows

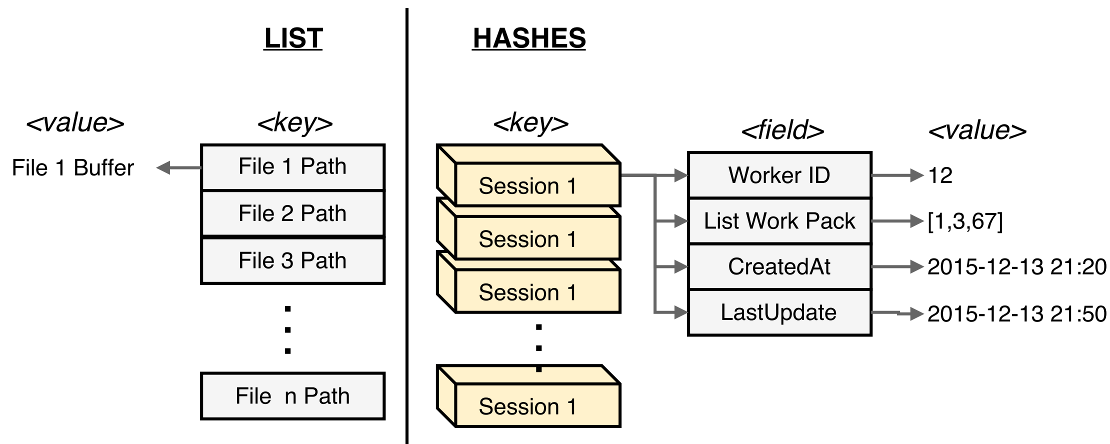


Figure 5.4: Redis data structures. Used for file storage and session management

how Promises can make the code easier to manage and understand. Every session has a determined time-to-live set at the time it is created (Line 24). Afterwards, it is automatically removed from the database.

Listing 13: Login method

```
1  /* Code omitted for brevity */
2
3  login: function (reqUsername, reqPassword) {
4
5    return Promise.resolve().then(function () {
6      return getUserFromDB(reqUsername); // return Worker
7    }).then(function (dbWorker) {
8      var hashPass = dbWorker.WK_HashPassword;
9      var salt = dbWorker.WK_Salt;
10
11     if (!validatePasswordHash("sha1", hashPass, reqPassword, salt)) {
12       throw loginMessages.loginFailed();
13     }
14     return dbWorker.WK_idWorker;
15   }).then(function (userID) {
16     var sessionKey = generateSessionKey();
17     var currentTime = timeUtils.currentTime();
18
19     sessionDB.hmsetAsync(sessionKey,
20       'worker', userID,
21       'createdAt', currentTime
22       'lastUpdated', currentTime
23     ).then(function () {
24       redis.expire(sessionKey, 259200); // Time in seconds
25     }).catch(function (err) {
26       throw errorMessages.generalErrorMessage(err);
```

```
27      });
28      return loginMessages.loginSuccessful(sessionKey);
29    }).catch(function (msg) {
30      throw msg;
31    });
32  }
```

The structure of the database reserved for caching the files is a simple Line 6, we defined the following option `return_buffers = true`, in the client.

5.2.3 *Client Configuration*

Apart from the server-side configuration of our solution, there is also configurations on the client-side. These configurations define the classes responsible for managing the application, the HTTP requests. Moreover, like on the server-side, there are dependency packages used in the development of the application

5.2.3.1 *Package Declaration*

The same way Node uses npm, Android uses Gradle, a flexible open-source build autonomous system [128]. It follows the same ideas of Apache Maven, Apache Ant, and it is used to efficiently compile or build projects that have dependencies on libraries. This automation guarantees that the build is reproducible no matter the time or environment. Listing 14 represents the content of the android's project `build.gradle` file.

Listing 14: build.gradle

```
1  apply plugin: 'com.android.application'
2
3  android {
4     compileSdkVersion 23
5     buildToolsVersion "23.0.1"
6
7    defaultConfig {
8      applicationId "de.tuhh.pmt.ar.client_android_app"
9        minSdkVersion 14
10       targetSdkVersion 23
11       versionCode 1
12       versionName "1.0"
13    }
14    buildTypes {
15        release {
16            minifyEnabled false
17            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
18        }
19    }
20    useLibrary 'org.apache.http.legacy'
```

```
21 }
22 dependencies {
23   compile fileTree(include: ['*.jar'], dir: 'libs')
24   compile 'com.android.support:appcompat-v7:23.1.0'
25   compile 'com.squareup.retrofit:retrofit:2.0.0-beta2'
26   compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2'
27   compile 'com.google.code.gson:gson:2.3'
28 }
```

In Listing 14, we can find information related to the Android-specific building options, such as: the compilation target; version of the build tools; the minimum Android Software Development Kit (SDK) supported; and the dependencies of the project. In Lines 22 through 28, we can see the dependency declaration of the Retrofit library (Section 4.3.2), and all necessary renaming dependencies. .

There is one small detailed about this build that is worth exploring. ProGuard, Line 17, is a tool that reduces the application's size by removing unused code, logging, debugging, and testing code. It also offers protection by obfuscating code, renaming classes, fields, and methods with semantically obscure names. These measure will make it harder to reverse engineering the application. By removing verbose logging code in a background service, the developer of Proguard, Eric Lafortune, states that battery life can increase up to five times [129].

5.2.3.2 *Android HTTP Client Interface*

In agreement with the services defined in Table 3.6, both the server and the client should be open to communicate through the stated URI. Before, we saw how these were defined on the server side. On the client, we created an interface, that can be used through the application to invoke those services. Listings 15 depicts one of the methods of that interface.

Listing 15: Client HTTP interface call method. Three parameters: a query parameters, session; and two path parameters, one for the Work Package ID and the other for the selection method

```
1 /* Code omitted for brevity */
2
3 /* Get the List Parts JSON File of a Work Packages */
4 @GET("/wp/{id}/listParts/{method}")
5 retrofit.Call<List<Parts>> getWorkPackageListParts(
6   @Query("session") String session,
7     @Path("id") String idWP,
8     @Path("method") String selectionMethod);
```

We decided, as it can be seen, to send the session key as a Query parameter. We discussed this in Section 3.3.4, as well as implementation alternatives and practices.

The method depicted corresponds to the request for a Work Package List Part JSON file. The body of the response is mapped to a list of parts class object. We will see the use of a method later in Section 5.4.

5.3 ASSEMBLY INSTRUCTIONS UPDATES METHOD

One of the core functions of our system, is the ability to query for updates made to assembly instructions. These requests are sent periodically to the server, in case the assembly instructions, currently being assembled, are outdated. An `IntentService` is designed to handle long-running tasks, without affecting the application UI thread. This service is started by an Activity, but is not bound to it, meaning it is not affected by the activity's lifecycle. Once the `IntentService` starts, it handles each intent using a worker thread and stops itself when the work is completed. Our `IntentService` class can be seen in Listing 16. If there are any updates, a notification is displayed to the worker (Line 21). In the future, this notification can be replaced by an automatic request for the updated instructions, removing the human element from the equation.

Listing 16: Update Service

```
1  /* Code omitted for brevity */
2
3  public class UpdateService extends IntentService {
4      public UpdateService() {
5         super("UpdateService");
6      }
7
8  @Override
9  protected void onHandleIntent(Intent intent) {
10   Call<List<WorkPackages>> listUpdatedWP = servicesAPI.updates(app.getSession());
11
12   listUpdatedWP.enqueue(new Callback<List<WorkPackages>>() {
13     @Override
14     public void onResponse(Response<List<WorkPackages>> response,
15       Retrofit retrofit) {
16
17       if(!response.isSuccess()) { /* Code omitted for brevity */ }
18
19       if(isUpdates(response)) {
20         workPackagesList = (ArrayList<WorkPackages>) response.body();
21         sendNotification("New Updates");
22       }
23     }
24   }
25 }
26
27 private void sendNotification(String msg) {
28   mNotificationManager = (NotificationManager)
         this.getSystemService(Context.NOTIFICATION_SERVICE);
29
30   PendingIntent contentIntent = PendingIntent.getActivity(this, 0,
31     new Intent(this, MainActivity.class), 0);
32
```

```
33   NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this)
34     .setContentTitle(getString(R.string.doodle_alert))
35     .setStyle(new NotificationCompat.BigTextStyle()
36     .bigText(msg))
37     .setContentText(msg);
38
39     mBuilder.setContentIntent(contentIntent);
40     mNotificationManager.notify(NOTIFICATION_ID, mBuilder.build());
41   }
42   /* Code omitted for brevity */
43 }
```

In this fashion, we can create requests to the server, however we still need a way to execute this periodically at a specified interval. To achieve this, we can invoke the intent *via* the `AlarmManager`, which will fire a `BroadcastIntent`. The `BroadcastReceiver`, a subset of `BroadcastIntent` is defined in Listing 17.

Listing 17: AlarmManager

```
1 /* Code omitted for brevity */
2
3 // Triggered by the Alarm periodically
4 @Override
5 public void onReceive(Context context, Intent intent) {
6   Intent i = new Intent(context, UpdateService.class);
7   context.startService(i);
8 }
```

As such, the `IntentService` task is ready to be executed periodically. What remains is to declare it in an activity, so that the application knows when to start. After this, the intent will live in the background, independent of the activity. The ideal activity to execute the intent, should be the one after the login, since we require a successful session key to perform the requests. It is also possible to declare it after the first Work Packages have been transfered. Listing 18 shows how the function `scheduleAlarm()` looks like. Requests are set to be sent every hour, Line 13.

Listing 18: Alarm scheduler

```
1 /* Code omitted for brevity */
2
3 // Setup a recurring alarm half hour
4 public void scheduleAlarm() {
5   Intent intent = new Intent(getApplicationContext(), MyAlarmReceiver.class);
6
7   // Create a PendingIntent to be triggered when the alarm goes off
8   final PendingIntent pIntent = PendingIntent.getBroadcast(this, MyAlarmReceiver.REQUEST_CODE,
        intent, PendingIntent.FLAG_UPDATE_CURRENT);
9
```

```
10   long firstMillis = System.currentTimeMillis(); // alarm is set right away
11   AlarmManager alarm = (AlarmManager) this.getSystemService(Context.ALARM_SERVICE);
12
13   alarm.setInexactRepeating(AlarmManager.RTC_WAKEUP, firstMillis, AlarmManager.INTERVAL_HOUR,
         pIntent);
14 }
```

5.4 TRANSFER METHODS IMPLEMENTATION

In Section 3.6.3, we describe two possible methods, Condense Method and Multiple-Requests Method), to for the transfer of the assembly instructions from the server to the client. We concluded that make an educated decision of which one should be supported, both would need to be implemented and testes. Only after, we would have enough data to decide. This section will explain the implementation of each method. The analysis and comparison of the methods can be found in Section 5.5.

5.4.1 *File Encryption*

All the files sent to the application are encrypted under a content encryption key, a symmetric key known only by the server and a specific user. As a cipher algorithm, we decided to use AES-256, a symetric block cipher. Whit the variation of encrypting data into blocks of 256 bits, and CTR mode. For an evaluation of the different block cipher modes, we recommend [130] and [131].

Listing 19: Encrypt Function, with AES-256 block cipher.

```
1  /* Code omitted for brevity */
2
3  var crypto = require('crypto');
4  var algorithm = 'aes-256-ctr';
5
6  function encryptedFile(buffer, sessionKey){
7    var cipher = crypto.createCipher(algorithm, sessionKey)
8    var crypted = Buffer.concat([cipher.update(buffer),cipher.final()]);
9    return crypted;
10 }
```

5.4.2 *Condense Method Implementation*

The logic behind this method was descried previously in Figure 3.9. In short, a compressed file is sent to the client with all the files and the List of Parts (Listing 2). As a security measure, this file is encrypted under a content encryption key, the session key used to authenticate a worker's requests.

Listing 20: Condense Method Service

```
/* Code omitted for brevity */

router.get('/wp/:id/:selection', validateSessionReq, function (req, res) {

  var wpId = req.params.id;
  var selection = req.params.selection;
  var sessionKey = req.query.session;

  workPackageService.getWorkPackage(wpId, selection)
    .then(function (zipper) {
      var encryptedFile = encryptFile(zipper, sessionKey);

      res.contentType('application/zip');
      res.setHeader(
        'content-disposition',
        'attachment; filename=' + CompressFile);

      res.send(encryptedFile)

    }).catch(function (err) {
      res.status(err.status).send(err.payload);
    });
});
```

Afterwards, the response is sent to the client where it was must be decipher and decompress. In Listing 21, we can see the function responsible for handling the request and response, on the client side. Due to the long running nature of these function, in order not to block the UI thread, we must use `AsyncTask` (Line 3). As to not block or collide with other requests or operations, all requests are put in a queue (Line 18), and executed when possible. If the response is successful, we can begin to decrypt its content and process the files (Line 26).

Listing 21: Android condense method

```
/* Code omitted for brevity */

public class getWorkPackageCompress extends AsyncTask<Void, Void, Boolean> {

  @Override
  protected Boolean doInBackground(Void... params) {

  String selectionMethod = params[0];
  String workPackageID = params[1];

  AndroidApp app = (AndroidApp) getApplication();
  String session = app.getSession();

```

```
14    RestServicesAPI servicesAPIFiles = app.getServicesAPIFiles();
15
16      Call<ResponseBody> responseZip = servicesAPI.getWorkPackage(session, workPackageID,
          selectionMethod);
17
18      responseZip.enqueue(new retrofit.Callback<ResponseBody>() {
19        @Override
20          public void onResponse(retrofit.Response<ResponseBody> response, Retrofit retrofit) {
21            if (!response.isSuccess()) { /* Code omitted for brevity */ }
22
23            try {
24              ResponseBody body = response.body();
25              InputStream inputStream = body.byteStream();
26              ZipInputStream zipInputStream = new ZipInputStream(decompressFile(inputStream));
27
28              /* Code omitted for brevity */
29              });
30        /* Code omitted for brevity */
31      }
32    }
```

5.4.3 *Multiple-Requests Method Implementation*

The second method, consists in re-using the initial connection for all subsequent requests. As we pointed out, there is one big difference when compare to the previous method. This, requires the List of Parts (Listing 2) to be sent to the application prior to the request for the assembly instructions. Listing 22 shows the application's code for the method. Again, an `AsyncTask` is used to not block the UI thread. Inside it, two HTTP requests are made. The first for the List of Parts (Lines 18), where afterwards a loop invokes a HTTP request for each of the parts inside the list (Line 27). The connection is maintained open and is only close after the loop has terminated (Line 31). Each file received is decompress, and should be passed to other function of the application. For testing purposes, both this method and the previous simply read and discard the data.

Listing 22: Android Multi-Request method

```
1  /* Code omitted for brevity */
2
3  public class getWorkPackageMutil extends AsyncTask<Void, Void, Boolean> {
4
5    @Override
6    protected Boolean doInBackground(Void... params) {
7
8    String selectionMethod = params[0];
9    String workPackageID = params[1];
10
11   AndroidApp app = (AndroidApp) getApplication();
```

```
12  String session = app.getSession();
13
14  RestServicesAPI servicesAPIFiles = app.getServicesAPIFiles();
15
16    Call<List<Parts>> listParts = servicesAPI.getWorkPackageListParts(session, workPackageID,
        selectionMethod);
17
18    listParts.enqueue(new retrofit.Callback<List<Parts>>() {
19      @Override
20        public void onResponse(retrofit.Response<ResponseBody> response, Retrofit retrofit) {
21          if (!response.isSuccess()) { /* Code omitted for brevity */ }
22
23          try {
24            List<Parts> body = response.body();
25
26           for(Parts p : body) {
27             InputStream in = makeRequestFile(session, p.path); // Request for File
28          decompressFile(inputStream);
29          /* Code omitted for brevity */
30           }
31        response.body().close(); // Connection Closes
32            /* Code omitted for brevity */
33            });
34      /* Code omitted for brevity */
35    }
36   }
```

At the server-side, Express.js takes care of most of the boilerplate implementation. Since the canonical path of a file can be used as an identifier, we can eliminated one database lookup. However, to avoid an injection attack, the request parameter for the path is first sanitized (Line 7).Afterwards, the file is read, encrypted and sent to the client (Lines 11 and 13).

Listing 23: Multi-Request Method Service

```
1  /* Code omitted for brevity */
2
3  router.get('/wp/file/:loc', validateSessionReq, function (req, res, next) {
4    var session = req.query.session;
5      var reqPath = req.params.loc;
6
7    reqPath = cleanPath(reqPath);
8
9    var fullPath = path.join(__dirname, filesLocation, reqPath);
10
11   var encryptedFile = encryptFile(workpackageServices.getFile(fullPath), sessionKey);
12
13     res.send(encryptedFile);
14 });
```

To guarantee that the connection was not established between request we used a packet analyzer, Wireshark, to analyze the traffic between the application and the server. Figure 5.5 shows the packets exchange between the server and the client. It is possible to verify, that in between requests, there are no *FIN* packets exchange. These only occur, after all requests have been transmitted to the client. Thus, we successfully implemented these method in accordance with our specifications.

31	16.658895000	192.168.1.28	192.168.1.12	TCP	74	39422→3000 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=68558120 TSecr=0 WS=256
33	16.659229000	192.168.1.28	192.168.1.12	TCP	74	53303→3000 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=68558120 TSecr=0 WS=256
35	16.659718000	192.168.1.28	192.168.1.12	TCP	74	32895→3000 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=68558120 TSecr=0 WS=256
37	16.659896000	192.168.1.28	192.168.1.12	TCP	74	58103→3000 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=68558121 TSecr=0 WS=256
39	16.660263000	192.168.1.28	192.168.1.12	TCP	74	49050→3000 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=68558121 TSecr=0 WS=256
41	16.767596000	192.168.1.28	192.168.1.12	TCP	66	39422→3000 [ACK] Seq=1 Ack=1 Win=87808 Len=0 TSval=68558125 TSecr=274500934
42	16.767599000	192.168.1.28	192.168.1.12	TCP	66	53303→3000 [ACK] Seq=1 Ack=1 Win=87808 Len=0 TSval=68558125 TSecr=274500934
43	16.767708000	192.168.1.28	192.168.1.12	TCP	66	32895→3000 [ACK] Seq=1 Ack=1 Win=87808 Len=0 TSval=68558125 TSecr=274500934
47	16.768172000	192.168.1.28	192.168.1.12	HTTP	197	GET /test/met2/7.obj HTTP/1.1
48	16.768173000	192.168.1.28	192.168.1.12	HTTP	197	GET /test/met2/6.txt HTTP/1.1
51	16.768468000	192.168.1.28	192.168.1.12	HTTP	197	GET /test/met2/5.obj HTTP/1.1
53	16.770972000	192.168.1.28	192.168.1.12	TCP	66	58103→3000 [ACK] Seq=1 Ack=1 Win=87808 Len=0 TSval=68558136 TSecr=274500934
54	16.770976000	192.168.1.28	192.168.1.12	TCP	66	49050→3000 [ACK] Seq=1 Ack=1 Win=87808 Len=0 TSval=68558136 TSecr=274500935
57	16.772600000	192.168.1.28	192.168.1.12	HTTP	197	GET /test/met2/8.txt HTTP/1.1
59	16.772721000	192.168.1.28	192.168.1.12	HTTP	197	GET /test/met2/9.obj HTTP/1.1
64	16.785871000	192.168.1.28	192.168.1.12	TCP	66	39422→3000 [ACK] Seq=132 Ack=271 Win=88832 Len=0 TSval=68558137 TSecr=274501055
65	16.786769000	192.168.1.28	192.168.1.12	TCP	66	53303→3000 [ACK] Seq=132 Ack=271 Win=88832 Len=0 TSval=68558138 TSecr=274501056
68	16.787665000	192.168.1.28	192.168.1.12	TCP	66	32895→3000 [ACK] Seq=132 Ack=271 Win=88832 Len=0 TSval=68558138 TSecr=274501056
69	16.788738000	192.168.1.28	192.168.1.12	TCP	66	49050→3000 [ACK] Seq=132 Ack=271 Win=88832 Len=0 TSval=68558138 TSecr=274501057
70	16.789633000	192.168.1.28	192.168.1.12	TCP	66	58103→3000 [ACK] Seq=132 Ack=271 Win=88832 Len=0 TSval=68558138 TSecr=274501058
73	16.934267000	192.168.1.28	192.168.1.12	HTTP	198	GET /test/met2/10.txt HTTP/1.1
75	16.934531000	192.168.1.28	192.168.1.12	HTTP	198	GET /test/met2/12.txt HTTP/1.1
77	16.934893000	192.168.1.28	192.168.1.12	HTTP	198	GET /test/met2/14.txt HTTP/1.1
79	16.935016000	192.168.1.28	192.168.1.12	HTTP	198	GET /test/met2/13.obj HTTP/1.1
81	16.935347000	192.168.1.28	192.168.1.12	HTTP	198	GET /test/met2/11.obj HTTP/1.1
302	17.890640000	192.168.1.28	192.168.1.12	HTTP	198	GET /test/met2/87.txt HTTP/1.1
303	17.890643000	192.168.1.28	192.168.1.12	HTTP	198	GET /test/met2/89.obj HTTP/1.1
308	17.927834000	192.168.1.28	192.168.1.12	TCP	66	32895→3000 [ACK] Seq=2112 Ack=4336 Win=104960 Len=0 TSval=68558252 TSecr=274502119
309	17.936034000	192.168.1.28	192.168.1.12	TCP	66	53303→3000 [ACK] Seq=2244 Ack=4607 Win=105984 Len=0 TSval=68558253 TSecr=274502120
5397	137.796678000	192.168.1.28	192.168.1.12	TCP	66	39422→3000 [ACK] Seq=2112 Ack=4337 Win=104960 Len=0 TSval=68570239 TSecr=274620797
5403	137.937780000	192.168.1.28	192.168.1.12	TCP	66	32895→3000 [ACK] Seq=2112 Ack=4337 Win=104960 Len=0 TSval=68570253 TSecr=274620960
5405	137.940420000	192.168.1.28	192.168.1.12	TCP	66	53303→3000 [ACK] Seq=2244 Ack=4608 Win=105984 Len=0 TSval=68570253 TSecr=274620960
5420	138.306650000	192.168.1.28	192.168.1.12	TCP	66	49050→3000 [ACK] Seq=2112 Ack=4337 Win=104960 Len=0 TSval=68570290 TSecr=274621229
5421	138.306652000	192.168.1.28	192.168.1.12	TCP	66	58103→3000 [ACK] Seq=1980 Ack=4066 Win=103680 Len=0 TSval=68570290 TSecr=274621238
7344	317.769353000	192.168.1.28	192.168.1.12	TCP	66	49050→3000 [FIN, ACK] Seq=2112 Ack=4337 Win=104960 Len=0 TSval=68588232 TSecr=274621358
7345	317.769356000	192.168.1.28	192.168.1.12	TCP	66	58103→3000 [FIN, ACK] Seq=1980 Ack=4066 Win=103680 Len=0 TSval=68588232 TSecr=274621358
7346	317.769465000	192.168.1.28	192.168.1.12	TCP	66	39422→3000 [FIN, ACK] Seq=2112 Ack=4337 Win=104960 Len=0 TSval=68588232 TSecr=274620858
7350	317.897409000	192.168.1.28	192.168.1.12	TCP	66	32895→3000 [FIN, ACK] Seq=2112 Ack=4337 Win=104960 Len=0 TSval=68588249 TSecr=274620994
7352	317.898619000	192.168.1.28	192.168.1.12	TCP	66	53303→3000 [FIN, ACK] Seq=2244 Ack=4608 Win=105984 Len=0 TSval=68588249 TSecr=274620996

Figure 5.5: HTTP packets capture through Wireshark.

5.5 TEST RESULTS

This section pertains to the analysis of the overall system's performance. Both methods are tested using a number of different files and scenarios. Also, to determine the server's behavior to peak load conditions, we performed some loading tests.

5.5.1 *Transfer Methods Comparison*

In Section 3.6.3.3, we described the main differences between the two methods designed to handle the transfer of the assembly instructions. We concluded, that both had their merits and should be implemented and tested to determine to most suitable one. Also, as we mentioned in Section 3.6.4.2, we tried to improved the response time by having files stored in memory.

The tests consisted in having an application transfer different Work Packages, each one with a different number of files and file's sizes. The tests were conducted throughout a period of time, and the results averaged. These results, measured in seconds, can be seen in Figure 5.6 and Table 5.3.

At first glance, we can see which of the two methods is the fastest, the Multiple-Requests Method. The reason why is understandable, compression is a very expensive process, and as we mentioned before, HTTP already supports *gZip* compression in the transmission, thus having two compression methods is redundant. Of course, that is not to say, that there could be times, in which having a compress file sent to a client can be useful. However, that is not our case, in fact, having the ability to perform a request for the exact file we need can increase the application's flexibility, specially when a updates have been made to a single part.

In regards to having files stored in memory, as expected, we can clearly see an improved in the response time when combined with both methods. Surely there are still improvements that can be made, nevertheless we believe we arrived at a good starting point for seamlessly transferring assembly instructions to the client application.

Table 5.3: Method results values

		Methods			
		Compress	**Compress + Cache**	**Multi-Request**	**Multi-Request + Cache**
Number of Files	*1*	0,2857	0,1664	0,1014	0,0364
	2	0,1571	0,1035	0,0714	0,0393
	5	0,3178	0,2223	0,1692	0,1162
	10	0,2264	0,1501	0,1068	0,0634
	20	0,2643	0,1641	0,1086	0,0532
	50	0,6213	0,3508	0,2103	0,0698
	80	0,9477	0,5262	0,3101	0,0941

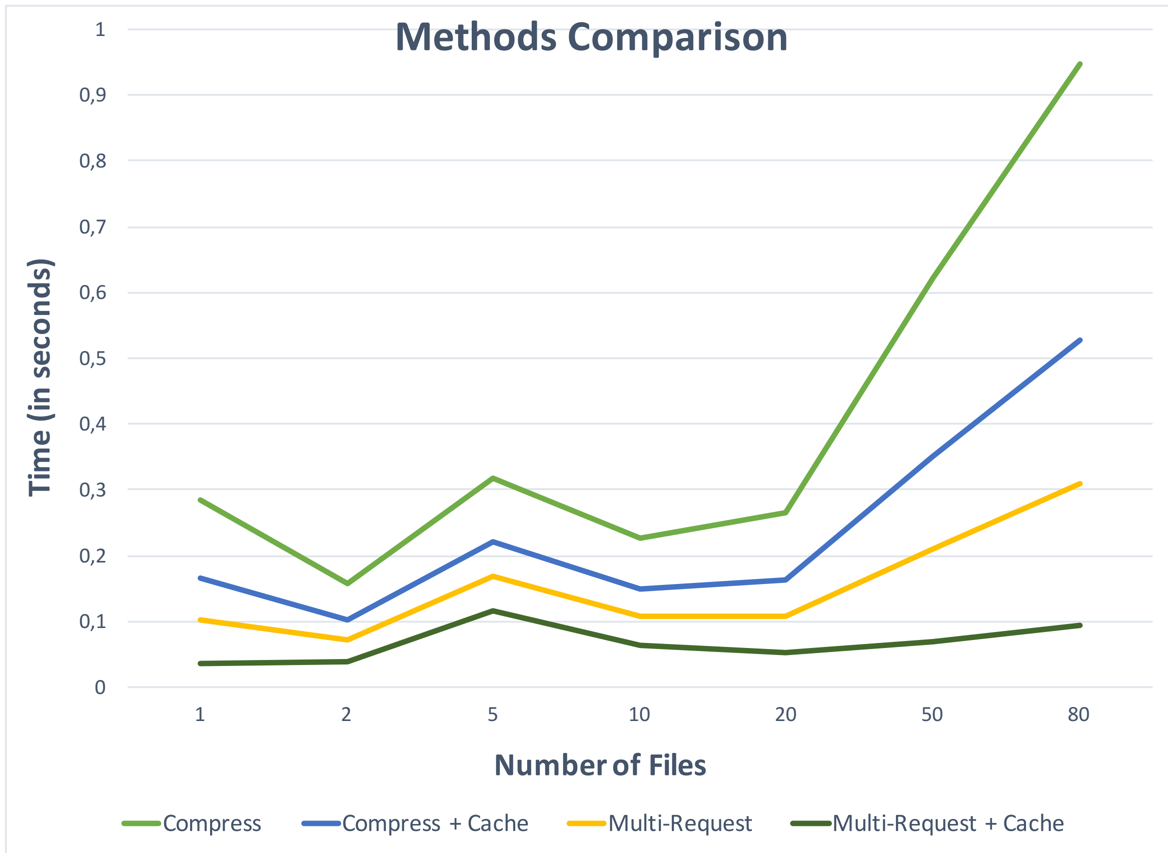


Figure 5.6: Methods comparison

5.5.2 *Load Testing*

There are times when we can expect a large number of requests been made to the server. When working begin their working day or shift, there will be an extreme amount of pressure to our server. In preparation for those occasions, we decided to perform a load testing scenario, to stress test our server. Multiple concurrent connections simulate multiple workers making requests to the server.

Assuming an assembly instruction is around 100 KB, and that an entire ship is comprised of 1000 separated files. For this test, we used Apache's JMeter[13] load testing tool, and simulated 100 workers concurrently requesting the entire ship's assembly instructions.

The results of the stress test can be seen in Figure 5.7 and Table 5.4. In 10 minutes, using cache, all workers could have the entire assembly instructions to build the ship. 14 minutes, in case the files were not stored in memory. These are fantastic numbers when compared

13 JMeter — http://jmeter.apache.org/

to the initial method of manually transferring the assembly instructions. More so, when we consider the impact it can have on the shipyard's productivity.

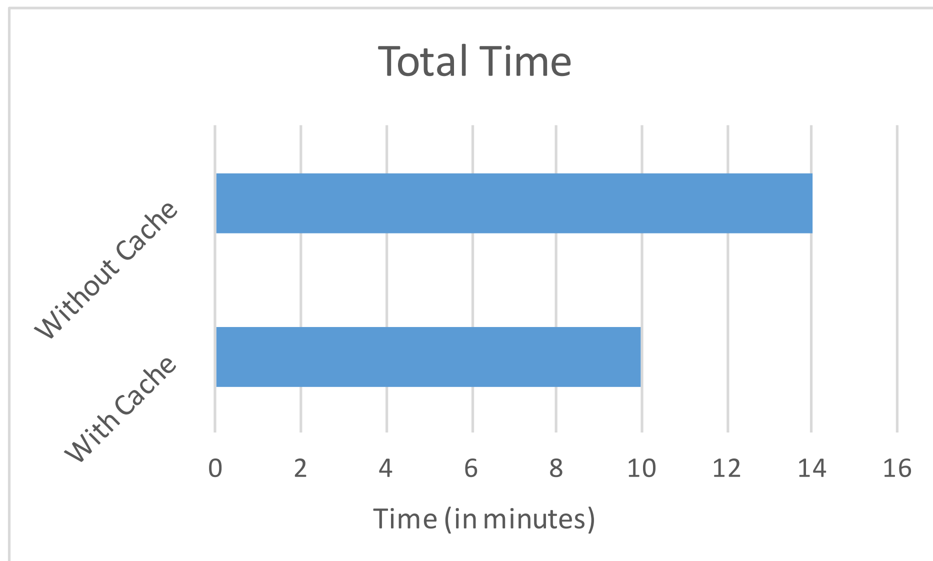


Figure 5.7: Load test total time.

Table 5.4: Load test summary results

	Without Cache	With Cache
# Samples	100000	100000
Average	354	409
Min	4	1
Max	1008	1092
Std. Dev.	153.39	109.02
Error %	0.00%	0.00%
Throughput	248.2	213.5
KB/sec	19801.95	17034.10
Avg. Bytes	81709.4	81711.0
Time (min)	14	10

In Table 5.4, we can analyze the summary results of both load test, the main terms are :

SAMPLES Number of HTTP requests ran for given thread. Since we simulated 100 workers requesting 1000 files, we have a total of 100000 samples.

AVERAGE Average response time for a particular HTTP request, measured in milliseconds.

MIN Min denotes the minimum response time of a HTTP request.

MAX Max denotes the maximum response time taken by the HTTP request.

STD.DEVIATION Number of exceptional cases found that deviated from the average value of the response time. The lesser this value the more consistent the time is.

ERROR % Number of HTTP requests that resulted in an error, i.e., were not successful. In this case, an error can be a `404 NOT FOUND`. In both tests, the percentage of errors is zero, meaning all requests ran successfully.

THROUGHPUT Number of requests, per unit of time (seconds, minutes, hours), that are sent to your server during the test.

6 CONCLUSION

During this project, we overviewed the shipbuilding assembly and the planning process behind it. With the knowledge collected from this revision, we were able to create an organizational relational model capable of holding all assembly instructions relevant to the ship's construction.

We started by introducing our concept of Work Packages, a bundle of all the components necessary for a specific task a worker must complete at any stage of construction. This was followed by the development of an automated assembly instructions selection protocol and a transfer server. The aim was allowing workers to easily access all the information via a user-friendly mobile application, increasing the efficiency and, consequently, the productivity.

The new model had to fulfill some critical requirements. First of all, it should be simple and easy to understand. However, regardless of its simplicity, the structure should be complex enough to hold all the information in an organized and intercorrelated way, between the worker, component, work package, and data. Finally, the model should be "universal", meaning able to be implemented in a different industry or to a different problem if necessary, without difficulty.

In this thesis, we defined three main points for the creation of transferring mechanisms: **(i)** communication technologies; **(ii)** transfer speed; **(iii)** and security. The first point relates to the technologies present today in most modern mobile devices. To take advantage of the devices portability, the server and the application should communicate wirelessly. We performed a study of different wireless communication and concluded Wi-Fi was the most suitable, due to its low implementation costs, fast transfer speeds, and wide range. Moreover, Wi-Fi antennas are standard equipment in the majority of mobile devices in the market, so no additional costs would be added. In the stack,

The second point refers to technologies and protocols used to establish a connection between the server and the application and transport the assembly instructions from one end to the other properly. Unlike their wired counterparts, mobile devices are subjected to severe limitations on power consumption, performance, size, and weight, all of which were taken into consideration when developing the transferring mechanisms. We demonstrated that TCP and HTTP, combined, have all the characteristics essential to guarantee a good performance to battery life ratio.

The third and final point concerns the security of both the connection, the data in transit, and data protection in the application. We explained how the use of SSL can ensure a private and secure connection. Moreover, every transferred data is encrypted under a symmetric content encryption key, known only by the server and the client. For added pro-

tection, we ensured that all credentials, necessary for the user and request authentication, are protected against exposition.

Two different transference methods were designed, developed, and tested in a series of scenarios varying in the size and nature of the files transferred. We concluded that reusing a TCP connection and performing all necessary requests over it, was the most efficient method. Moreover, we successfully implemented two additional techniques to increase the response time. Stress/Load tests were also performed to assess the server behavior under pressure in normal and anticipated peak load conditions. Due to the unavailability of a real life environment, we were incapable of performing reliability tests. However, during implementation, measures were taken to ensure that a worker always has the most updated information. Nevertheless, these tests should be conducted in the future.

Besides easing the distribution of assembly instructions between workers, it was demonstrated how the introduction of this work can contribute to the rise of ship's construction productivity, especially when compared to the previously assembly instructions manual transfer. The initial context for this thesis is the assembly instructions for the marine industry, however, due to its abstract nature and modular design, it can be easily implemented in other industries.

We were able to successfully fulfill the objectives established for this project. Moreover, in pursuit of a more complete solution, an effort was made to develop extra components and improve the design. Further improvements can be added in other areas of this solution. In the future, in order to enhance the file memory storage we can incorporate prediction algorithms that anticipate a request for a file. This could be achieved either by logging in worker activity and predict its habits, or through the shipyards' time attendance system (punch clock). Of course, there is a high complexity associated with these cases if the business logic allows workers to freely choose Work Packages. Another improvement worth considering is the development of a Graphical User Interface (GUI), for the management of Work Packages, allowing foremen or project managers to watch over the state and progress of the construction. Gantt charts are a practical tool for illustrating a project schedule, a feasible tool considering the availability of time attributes associated with Work Packages.

BIBLIOGRAPHY

[1] Hermann Lodding Philipp Sebastian Halata, Axel Friedewald. Augmented reality supported information gathering in one-of-a-kind production. In *13th International Conference on Computer and IT Applications in the Maritime Industries (COMPIT '14)*, pages 489–503. Redworth, 2014.

[2] Klaas Dokkum and Carmen Koenen-Loos. *Ship Knowledge: Ship Design, Construction and Operation*. DOKMAR, 3th edition, 2006.

[3] Marine Exchange of Alaska (MXAK). URL http://www.mxak.org/community/polar_adventure/side%20view.jpg. [Online; accessed 28-November-2015].

[4] D.J. Eyres and G.J. Bruce. 12 - Design information for production. In D.J. EyresG.J. Bruce, editor, *Ship Construction (Seventh Edition)*, pages 125 – 134. Butterworth-Heinemann, Oxford, seventh edition edition, 2012. ISBN 978-0-08-097239-8.

[5] Eric C. Tupper. Chapter 14 - ship design. In Eric C. Tupper, editor, *Introduction to Naval Architecture (Fifth Edition)*, pages 343 – 377. Butterworth-Heinemann, Oxford, fifth edition edition, 2013. ISBN 978-0-08-098237-3.

[6] Dictionary.com — Bert, Nov 2015. URL http://dictionary.reference.com/browse/berth. [Online; accessed 04-November-2015].

[7] Dictionary.com — Keel. Nov 2015. URL http://dictionary.reference.com/browse/keel. [Online; accessed 04-November-2015].

[8] J.F.C. Conn. Ship construction. *Encyclopedia Britannica*, 2015. URL http://www.britannica.com/technology/ship-construction. [Online; accessed 04-November-2015].

[9] Paul Stott. *UFPE Recife: Shipbuilding Competitiveness — Work breakdown and competitiveness*. Newcastle University, 2013.

[10] J-D Caprace, Cristian Petcu, MG Velarde, and Philippe Rigo. Optimization of shipyard space allocation and scheduling using a heuristic algorithm. *Journal of marine science and technology*, 18(3):404–417, 2013.

[11] H Kim, S-S Lee, JH Park, and J-G Lee. A model for a simulation-based shipbuilding system in a shipyard manufacturing process. *International Journal of Computer Integrated Manufacturing*, 18(6):427–441, 2005.

[12] Bath Iron Works Corporation, Corporate-Tech Planning, United States. Maritime Administration, and National Shipbuilding Research Program. *A Manual on Planning and Production Control for Shipyard Use*. The Corporation, 1978.

[13] Howard M Bunch. A study of the construction planning and manpower schedules for building the multipurpose mobilization ship, pd214, in a shipyard of the people's republic of china. *Journal of ship production*, (4), 1988.

[14] Sylvia Encheva, Sharil Tumin, and Maryna Z Solesvik. Decision support system for assessing participants reliabilities in shipbuilding. In *Proceedings of the 9th WSEAS international conference on Automatic control, modelling and simulation*, pages 270–275. World Scientific and Engineering Academy and Society (WSEAS), 2007.

[15] A.G. Greenwood, S. Vanguri, B. Eksioglu, P. Jain, T.W. Hill, J.W. Miller, and C.T. Walden. Simulation optimization decision support system for ship panel shop operations. In *Simulation Conference, 2005 Proceedings of the Winter*, pages 9 pp.–, Dec 2005.

[16] G. Chondrocoukis and E. Foundas. A group decision support system design for a shiprepair facility. *Journal of Information and Optimization Sciences*, 18(2):261–269, 1997.

[17] John Earley. Computer Weekly — Infrared meets speed and security needs, 2015. URL http://www.computerweekly.com/opinion/Infrared-meets-speed-and-security-needs. [Online; accessed 07-November-2015].

[18] Bradley Mitchell. About Tech — Infrared, . URL http://compnetworking.about.com/od/homenetworking/g/bldef_infrared.htm. [Online; accessed 07-November-2015].

[19] Darren Quick. Gizmag — Infrared technology offers faster wireless data transfer than wi-fi and bluetooth, October 2012. URL http://www.gizmag.com/infrared-optical-wireless-data-module/24373/. [Online; accessed 07-November-2015].

[20] Bluetooth sig. URL http://www.bluetooth.com/what-is-bluetooth-technology/bluetooth. [Online; accessed 07-November-2015].

[21] Ian Paul. PC World — Wi-Fi direct vs. bluetooth 4.0: A battle for supremacy. URL http://www.pcworld.com/article/208778/Wi_Fi_Direct_vs_Bluetooth_4_0_A_Battle_for_Supremacy.html. [Online; accessed 07-November-2015].

[22] Bradley Mitchell. About Tech — Bluetooth, . URL http://compnetworking.about.com/cs/bluetooth/g/bldef_bluetooth.htm. [Online; accessed 07-November-2015].

[23] Vangie Beal. Webopedia — wi-fi. URL http://www.webopedia.com/TERM/W/Wi_Fi.html. [Online; accessed 23-November-2015].

[24] Richard Van Nee. Breaking the gigabit-per-second barrier with 802.11 ac. *Wireless Communications, IEEE*, 18(2):4–4, 2011.

[25] Bradley Mitchell. About Tech — wireless standards 802.11a, 802.11b/g/n, and 802.11ac. the 802.11 family explained, . URL http://compnetworking.about.com/cs/wireless80211/a/aa80211standard.htm. [Online; accessed 24-November-2015].

[26] Home Network Admin. Wireless b vs g vs n vs ac | what is the difference? URL http://homenetworkadmin.com/wireless-b-vs-g-vs-n-vs-ac-difference/. [Online; accessed 24-November-2015].

[27] Aeroflex David Asquith. Wide bandwidth measurement techniques for 802.11ac wlan devices. *Microwave Product Digest*, August 2012. URL http://www.mpdigest.com/issue/Articles/2012/Aug/Aeroflex/Default.asp. [Online; accessed 24-November-2015].

[28] AT&T. Differences between 802.11a, 802.11b, 802.11g and 802.11n. URL https://www.wireless.att.com/support_static_files/KB/KB3895.html. [Online; accessed 24-November-2015].

[29] Ajay R Mishra. *Fundamentals of cellular network planning and optimisation: 2G/2.5 G/3G... evolution to 4G*. John Wiley & Sons, 2004.

[30] Ofcom. Measuring mobile broadband performance in the uk. Technical report, November 2014.

[31] Michael Miller. *Wireless Networking Absolute Beginner's Guide*. Que Publishing, 2013.

[32] Jessica Rosenworcel. Growing unlicensed spectrum, growing the wireless economy. URL http://recode.net/2014/02/21/growing-unlicensed-spectrum-growing-the-wireless-economy/. [Online; accessed 24-November-2015].

[33] Gian Paolo Perrucci, Frank HP Fitzek, and Jörg Widmer. Survey on energy consumption entities on the smartphone platform. In *Vehicular Technology Conference (VTC Spring), 2011 IEEE 73rd*, pages 1–6. IEEE, 2011.

[34] Eric Rozner, Vishnu Navda, Ramachandran Ramjee, and Shravan Rayanchu. Napman: network-assisted power management for wifi devices. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys 10, pages 91–106. ACM, 2010.

[35] R Nave. Inverse square law, general. URL http://hyperphysics.phy-astr.gsu.edu/hbase/forces/isq.html. [Online; accessed 24-November-2015].

[36] Emmett Dulaney and Michael Harwood. *CompTIA Network+ N10-005 Exam Cram*. Pearson IT Certification, 4 edition, January 2012.

[37] David Callisch. Coping with wi-fi's biggest problem: interference, August 2010. URL http://www.networkworld.com/article/2215287/tech-primers/coping-with-wi-fi-s-biggest-problem--interference.html. [Online; accessed 01-December-2015].

[38] Charles Kozierok. The tcp/ip guide: A comprehensive, illustrated internet protocols reference, September 2005.

[39] Keith Winstein. Forbes tech | how does one decide between tcp and udp?, January 2014. URL http://www.forbes.com/sites/quora/2014/01/27/how-does-one-decide-between-tcp-and-udp/. [Online; accessed 01-December-2015].

[40] Jakob Nielsen. Hypertext and hypermedia. 1990.

[41] R Fielding, James Gettys, Jeffrey C Mogul, H Frystyk, Larry Masinter, P Leach, and T Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Network Working Group, June 1999.

[42] Can i use web sockets? URL http://caniuse.com/#feat=websockets. [Online; accessed 01-December-2015].

[43] P Willars. Smartphone traffic impact on battery and networks. October 2010. URL http://www.ericsson.com/research-blog/uncategorized/smartphone-traffic-impact-battery-networks/. [Online; accessed 03-December-2015].

[44] Engin Bozdag, Ali Mesbah, and Arie Van Deursen. A comparison of push and pull techniques for ajax. In *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on*, pages 15–22. IEEE, 2007.

[45] Sharon Shea Margaret Rouse. Access control definition, June 2014. URL http://searchsecurity.techtarget.com/definition/access-control. [Online; accessed 14-November-2015].

[46] Duke University. Authentication vs. authorization, October 2012. URL http://www.duke.edu/~rob/kerberos/authvauth.html. [Online; accessed 14-November-2015].

[47] Federal Financial Institutions Examination Council. Authentication in an internet banking environment. *Financial Institution Letter, FIL-103-2005. Washington, DC: Federal Deposit Insurance Corp.(FDIC). Retrieved March*, 18:2005, 2005.

[48] Bruce Schneier. Two-factor authentication: too little, too late. *Commun. ACM*, 48(4): 136, 2005.

[49] Kirk Hausman, Martin Weiss, and Diane Barrett. *CompTIA Security+ SY0-301 Exam Cram*. Pearson Education, 2011.

[50] John Brainard, Ari Juels, Ronald L. Rivest, Michael Szydlo, and Moti Yung. Fourth-factor authentication: Somebody you know. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 168–178. ACM, 2006. ISBN 1-59593-518-5.

[51] Paul J Leach, John Franks, Ari Luotonen, Phillip M Hallam-Baker, Scott D Lawrence, Jeffery L Hostetler, and Lawrence C Stewart. Http authentication: Basic and digest access authentication. 1999.

[52] David F Ferraiolo and D Richard Kuhn. Role-based access controls. *arXiv preprint arXiv:0903.2171*, 2009.

[53] Oracle. Understanding web service security concepts. URL https://docs.oracle.com/middleware/1212/owsm/OWSMC/owsm-security-concepts.htm. [Online; accessed 16-November-2015].

[54] Min-kyu Choi, R Robles, Chang-hwa Hong, and Tai-hoon Kim. Wireless network security: Vulnerabilities, threats and countermeasures. *International journal of Multimedia and Ubiquitous Engineering*, 3(3), July 2008.

[55] Raymond Panko Reviews, C.T. *e-Study Guide for: Corporate Computer and Network Security*. Cram101, 2014. ISBN 9781467262279.

[56] F. Garzia. *Handbook of Communications Security*. WIT Press, 2013. ISBN 9781845647681.

[57] Edney and William A. Arbaugh. *Real 802.11 Security: Wi-Fi Protected Access and 802.11I*. Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321136209.

[58] Mark Ciampa. *CWNA guide to wireless LANs*. Cengage Learning, 2012. [Accessed 26-November-2015].

[59] Digitcert. What is ssl (secure sockets layer) and what are ssl certificates? URL https://www.digicert.com/ssl.htm. [Online; accessed 8-November-2015].

[60] Tim Dierks. The transport layer security (tls) protocol version 1.2. 2008.

[61] Holly Lynne McKinley. Ssl and tls: A beginners' guide. *SANS Institute*, 2003.

[62] Alan Freier, Philip Karlton, and Paul Kocher. The secure sockets layer (ssl) protocol version 3.0. 2011.

[63] Joseph Salowey. Transport layer security (tls) session resumption without server-side state. *Transport*, 2008.

[64] A Langley, N Modadugu, and B Moeller. Transport layer security (tls) false start. *draft-bmoeller-tls-falsestart-00, June, 2, 2010.* URL https://tools.ietf.org/html/draft-ietf-tls-falsestart-01. [Online; accessed 03-December-2015].

[65] Open Web Application Security Project (OWASP). Glossary: Certificate authority (ca), . URL https://www.owasp.org/index.php?title=Glossary&oldid=194291#Certification_Authority. [Online; accessed 03-December-2015].

[66] S Renfro. Secure browsing by default. *Facebook Engineering*, 571397832 (0), 2013. URL https://www.facebook.com/notes/facebook-engineering/secure-browsing-by-default/10151590414803920. [Online; accessed 26-November-2015].

[67] Michal Zalewski. Browser security handbook. *Google Code*, 2010. URL https://code.google.com/p/browsersec/wiki/Main. [Online, Accessed 26-November-2015].

[68] Adam Langley, N Modadugu, and WT Chang. Overclocking ssl. In *Velocity: Web Performance and Operations Conference*, 2010.

[69] SR Subramanya and Byung K Yi. Digital rights management. *Potentials, IEEE*, 25(2): 31–34, 2006.

[70] Daniel Nye Griffiths. The Truth Is, It Doesn't Work' - CD Projekt On DRM. *Forbes | Tech*, May 2012. URL http://www.forbes.com/sites/danielnyegriffiths/2012/05/18/the-truth-is-it-doesnt-work-cd-projekt-on-drm/. [Online; accessed 17-November-2015].

[71] Paul Hyman. PC Game Piracy: Why Bother With DRM? URL http://www.gamasutra.com/view/feature/132411/pc_game_piracy_why_bother_with_.php?print=1. [Online; accessed 17-November-2015].

[72] Michael A Einhorn and Bill Rosenblatt. Peer-to-peer networking and digital rights management: How market tools can solve copyright problems. *J. Copyright Soc'y USA*, 52:239, 2004.

[73] Amazon Web Services. Signing and authenticating rest requests. URL http://docs.aws.amazon.com/AmazonS3/latest/dev/RESTAuthentication.html. [Online; accessed 03-December-2015].

[74] Jeremy Kirk. Gawker media hacked, warns users to change passwords, December 2010. URL http://www.networkworld.com/article/2196643/malware-cybercrime/gawker-media-hacked--warns-users-to-change-passwords.html. [Online; accessed 03-December-2015].

[75] Zack Whittaker. Amazon force-resets some account passwords, citing password leak, November 2014. URL http://www.zdnet.com/article/amazon-is-resetting-account-passwords-for-some-accounts/. [Online; accessed 03-December-2015].

[76] Osmar R Zaïane and K Koperski. Cmpt 354 database systems and structures, 1998. URL http://www.cs.sfu.ca/CourseCentral/354/zaiane/material/notes/contents.html. [Online; accessed 25-November-2015].

[77] Thomas Fox-Brewster. 13 million passwords appear to have leaked from this free web host. Forbes, October 2015. URL http://www.forbes.com/sites/thomasbrewster/2015/10/28/000webhost-database-leak/. [Online; accessed 08-December-2015].

[78] Open Web Application Security Project (OWASP). Password storage cheat sheet, . URL https://www.owasp.org/index.php?title=Password_Storage_Cheat_Sheet&oldid=203402. [Online; accessed 08-December-2015].

[79] Kevin Wall John Steven, Jeff Walton. Secure password storage, September 2012. URL http://goo.gl/Spvzs. [Online; accessed 08-December-2015].

[80] D Eastlake 3rd and Paul Jones. Us secure hash algorithm 1 (sha1). Technical report, September 2001.

[81] Chrysanthou Yiannis. Modern password cracking: A hands-on approach to creating an optimised and versatile attack. 2013.

[82] Defuse Security. Salted password hashing - doing it right, August 2014. URL https://crackstation.net/hashing-security.htm. [Online; accessed 08-December-2015].

[83] Wenjing Zhou, Xiangwei Xie, Hui Li, Xiao Zhang, and Shan Wang. A database approach for accelerate video data access. In *Advances in Web and Network Technologies, and Information Management*, pages 45–57. Springer, 2009.

[84] Russell Sears, Catharine Van Ingen, and Jim Gray. To blob or not to blob: Large object storage in a database or a filesystem? *arXiv preprint cs/0701168*, April 2006.

[85] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010. ISBN 9780080553849.

[86] Mark Donald Hill, Norman Paul Jouppi, and Gurindar Sohi. *Readings in computer architecture*. Gulf Professional Publishing, 2000. [Accessed 25-November-2015].

[87] World Wide Web Consortium et al. Extensible markup language (xml) 1.1. 2006.

[88] Tim Bray. The javascript object notation (json) data interchange format. 2014.

[89] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: A case study. *Caine*, 9:157–162, 2009.

[90] Guanhua Wang. Improving data transmission in web applications via the translation between xml and json. In *Communications and Mobile Computing (CMC), 2011 Third International Conference on*, pages 182–185. IEEE, 2011.

[91] Yan Wei and Ubald Nienhuis. Automatic generation of assembly sequence for the planning of outfitting processes in shipbuilding. *Journal of Ship Production and Design*, 28(2):49–59, 2012.

[92] xyzws. What is the difference between absolute, relative and canonical path of file or directory? URL http://www.xyzws.com/Javafaq/what-is-the-difference-between-absolute-relative-and-canonical-path-of-file-or-directory/60. [Online; accessed 9-December-2015].

[93] Hong-Tai Chou and David J DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(1-4):311–336, 1986.

[94] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 5 byte rule for trading memory for cpu time. In *ACM SIGMOD Record*, volume 16, pages 395–398. ACM, 1986.

[95] RedisLabs. Introduction to redis. URL http://redis.io/topics/introduction. [Online; accessed 07-December-2015].

[96] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.

[97] Node.js Foundation. Node.js foundation, . URL https://nodejs.org/en/foundation/. [Online; accessed 07-December-2015].

[98] Linux Foundation. Linux foundation collaborative projects, . URL http://collabprojects.linuxfoundation.org/. [Online; accessed 07-December-2015].

[99] Node.js Foundation. About node.js, . URL https://nodejs.org/en/about/. [Online; accessed 07-December-2015].

[100] BEA Systems. Advantages and disadvantages of a multithreaded/multicontexted application. URL https://docs.oracle.com/cd/E13203_01/tuxedo/tux71/html/pgthr5.htm. [Online; accessed 07-December-2015].

[101] C.J. Ihrig. *Pro Node.js for Developers*. Expert's voice in Web development. Apress, 2013. ISBN 9781430258612.

[102] Ryan Dahl. Jsconf eu: Original node.js presentation, 2009. URL https://www.youtube.com/watch?v=ztspvPYybIY. [Online; accessed 07-December-2015].

[103] Jason Hoffman. Devcon5 santa clara: Node.js in context with jason hoffman of joyent, 2012. URL https://www.youtube.com/watch?v=yluchvyUzvU. [Online; accessed 07-December-2015].

[104] StrongLoop. Express.js. URL http://expressjs.com/en/index.html. [Online; accessed 07-December-2015].

[105] E. Brown. *Web Development with Node and Express: Leveraging the JavaScript Stack*. O'Reilly Media, 2014. ISBN 9781491902301.

[106] A. Mardan. *Express.js Guide: The Comprehensive Book on Express.js*. Azat Mardan, 2014.

[107] Sequelize. URL http://docs.sequelizejs.com/en/latest/. [Online; accessed 07-December-2015].

[108] Node-hat. URL https://github.com/substack/node-hat. [Online; accessed 07-December-2015].

[109] Bluebird. URL http://bluebirdjs.com/docs/getting-started.html. [Online; accessed 07-December-2015].

[110] M Ogden. Callback hell, 2015. URL http://callbackhell.com/. [Online; accessed 07-December-2015].

[111] Jake Archibald. Javascript promises: There and back again. URL http://www.html5rocks.com/en/tutorials/es6/promises/. [Online; accessed 07-December-2015].

[112] Mozilla Developer Network (MDN). Promise. URL https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Online; accessed 07-December-2015].

[113] Daniel Parker. *JavaScript with Promises*. " O'Reilly Media, Inc.", 2015.

[114] Chris Talkington. Archiver. URL https://www.npmjs.com/package/archiver. [Online; accessed 07-December-2015].

[115] Google Android Developers. Application fundamentals. 2009. URL http://developer.android.com/guide/components/fundamental.html. [Online; accessed 07-December-2015].

[116] Reto Meier. *Professional Android 4 application development*. John Wiley & Sons, 2012.

[117] Google Android Developers. Activity lifecycle. URL http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle. [Online; accessed 07-December-2015].

[118] Square. Retrofit. URL http://square.github.io/retrofit/. [Online; accessed 07-December-2015].

[119] Oracle. Datetime datatypes and time zone support. In *Oracle Database Globalization Support Guide*, chapter 4, page 11. 2015.

[120] CJ Date. Referential integrity. In *VLDB*, volume 81, pages 2–12, 1981. [Online; accessed 09-December-2015].

[121] Marco A Casanova and Luiz Tucherman. Enforcing inclusion dependencies and referencial integrity. 1988.

[122] Bill Karwin. *SQL antipatterns: avoiding the pitfalls of database programming*. Pragmatic Bookshelf, 2010. [Online; accessed 9-December-2015].

[123] Kari J Lieberherr and Ian M Holland. Assuring good style for object-oriented programs. *Software, IEEE*, 6(5):38–48, 1989.

[124] David Bock. The paperboy, the wallet, and the law of demeter, 2000.

[125] Tony Marston. What is the 3-tier architecture?, October 2012. URL http://www.tonymarston.net/php-mysql/3-tier-architecture.html. [Accessed 09-December-2015].

[126] Virinder Mohan Batra. Connection pool management for backend servers using common interface, August 15 2000. US Patent 6,105,067.

[127] Stephen T Kent. Internet privacy enhanced mail. *Communications of the ACM*, 36(8): 48–60, 1993.

[128] Tim Berglund and Matthew McCullough. *Building and Testing with Gradle*. " O'Reilly Media, Inc.", 2011.

[129] Eric Lafortune et al. Proguard, 2009. URL http://www.guardsquare.com/files/media/slides/ProGuard_IstanbulTechTalks2014.pdf. [Online; accessed 12-December-2015].

[130] Phillip Rogaway. Evaluation of some blockcipher modes of operation. *Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan*, 2011.

[131] Jakob Jonsson. On the security of ctr+ cbc-mac. In *selected Areas in Cryptography*, pages 76–93. Springer, 2003.

[132] Digital Command Control (DCC. Simplex, duplex and infrared, November 2014. URL http://www.dccwiki.com/index.php?title=Simplex,_Duplex_and_Infrared&oldid=11117. [Online; accessed 07-November-2015].

COMMUNICATION CHANNELS

There are three basic communication systems or mode of operation a channel can use to transmit information from one party to the other:

SIMPLEX In a simplex operation, the flow of information occurs only in one direction, meaning it is a "one-way street" — Figure .1. An example is a car's radio, which only receives and does not transmit [132].

HALF-DUPLEX Half-duplex operations are capable of sending information in both directions between two parties, but only in one direction at a time. "This is a fairly common mode of operation when there is only a single network medium (cable, radio frequency, and so forth) between devices". [38] In conventional networks, any device can transmit information, but only one can do so at a time — Figure .2.

FULL-DUPLEX Sometimes just called "Duplex" for redundancy, is when both parties can communicate simultaneous with one another. A good example of the operations is a telephone. Duplex channels can be compose of either a pair of simplex (as described above), or by using a channel that allows bidirectional communication simultaneously. [38] — Figure .3.

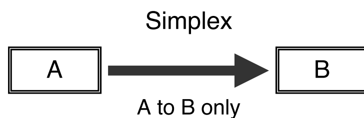


Figure .1: Simplex channel, A to B only.

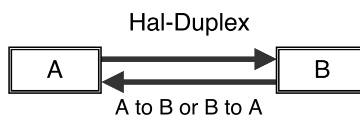


Figure .2: Half-Duplex channel, A to B or B to A.

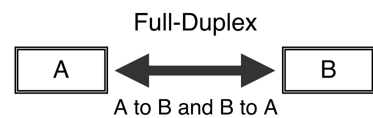


Figure .3: Full-Duplex channel, A to B and B to A.