# Modern Maritime Information Support Systems with Microservices Architecture: Designing, Developing, and Deploying

劉, 鴻澤

(Degree)
博士（工学）

(Date of Degree)
2023-03-25

(Date of Publication)
2024-03-01

(Resource Type)
doctoral thesis

(Report Number)
甲第8667号

(URL)
https://hdl.handle.net/20.500.14094/0100482415

Doctoral Dissertation

# Modern Maritime Information Support Systems with Microservices Architecture:

## Designing, Developing, and Deploying

# （マイクロサービス アーキテクチャによる 現代海事情報支援システム）

— 設計・開発・配置について —

January 2023

Graduate School of Maritime Sciences
Kobe University

LIU HONGZE
（ 劉　鴻澤 ）

DISSERTATION

# Modern Maritime Information Support Systems With Microservices Architecture

## Designing, Developing, and Deploying

LIU HONGZE

January 2023

Kobe University

**Disclaimer**
The views and opinions expressed in this dissertation are those of the author and do not necessarily reflect the official policy or position of Kobe University. The author assumes full responsibility for the accuracy and completeness of the information presented in this dissertation. The author makes no warranties, express or implied, with respect to the contents of this dissertation, including but not limited to any warranties of merchantability or fitness for a particular purpose. The author shall not be liable for damages of any kind arising from the use of this dissertation.

**Publisher**
Published by Kobe University

O EVER YOUTHFUL, O EVER WEEPING

– Jack Kerouac, *The Dharm Bums*

# Contents

Modern Communication Framework and Cross-Platform Applications 49

Modern Deployment Approach and Microservices Containerization 75

# List of Figures

# List of Tables

# Introduction 1

The maritime industry has played a critical role in shaping the modern world, connecting people and nations through trade and commerce. However, as the industry becomes more complex and competitive, the effective sharing, management, and application of maritime information have become increasingly important. Advances in technology have created new opportunities for the maritime industry to better manage this information and improve its operations. This chapter provides an introduction to this dissertation, including background and motivation, main ideas, and an outline of the dissertation.

## 1.1 Background and Motivation

The growth and development of the maritime industry have been a direct result of the continuous efforts of human navigation throughout history. From the early days of prehistoric Austronesian expansion [1] and Polynesian navigation [2] to the present day, humans have never ceased in their endeavors to traverse the seas and oceans, exploring new territories and establishing trade and commerce relationships with other civilizations. The development of trade and commerce, in turn, has led to the growth of the maritime industry, which today plays a critical role in the global economy.

The maritime industry encompasses a wide range of activities, from shipbuilding to cargo transportation, fishing, and tourism. Among these, the shipping industry occupies an important place, transporting raw materials and finished goods from country to country, accounting for more than 80% of world trade by volume and more than 70% by value [3, 4]. The maritime industry plays a vital role in facilitating international trade and commerce, thereby fostering economic growth and development worldwide.

To operate efficiently and effectively in this complex and ever-evolving industry, it is essential to have access to accurate and timely (sometimes historical) maritime information, which typically includes information on shipping lanes, ports, weather conditions, cargo and vessel movements, *etc.* Maritime information is critical to the industry to ensure the safe and efficient operation of vessels, plan and execute trade routes, mitigate risk, *etc.* As a result, maritime communication technologies are becoming increasingly important for sharing this information.

It is said that the earliest Polynesian navigators did not use any on-board equipment to aid navigation, relying solely on visual observation of birds and stars [5]. However, as human civilization advanced, so did the methods and tools of navigation and marine communication. Hundreds of years ago, humans evolved to use navigational instruments such as compasses and sextants to aid in positioning [6], and to use natural signals such as sound and light to communicate in limited ways with steam whistles and signal flags [7, 8].

The technological revolution, also known as the Third Industrial Revolution, which began in the mid-20th century and continues today, has enabled human civilization to take another giant step forward, moving humanity from the era of mechanical and analog electronics to the era of digital and information technology [9]. In just a few decades, it has brought about a sea change — technologies have shifted from analog computers to digital computers (1950s), from telegraphs to faxes (1980s), from analog telephones to digital mobile phones (1990s), and further to smartphones (2010s), and so on. Among them, digital transmission technology is considered to be one of the pillars. The emergence and rapid development of computer networks, digital broadcasting, various radio technologies, *etc.*, and especially the internet, have brought communication into an unprecedented new era.

With the rapid advancement of digital technology, the maritime industry is not immune to the trend of digitalization. Maritime information digitalization refers to the process of converting traditional maritime information into digital format and integrating it into a comprehensive information management system [10]. Through the digitalization, the maritime industry can significantly improve its operational efficiency, enhance monitoring, control, quality assurance and verification, and strengthen decision-making capabilities by providing a clearer and more accurate understanding of maritime conditions [11]. This is in stark contrast to traditional maritime communication systems that rely on radio waves [12], such as Morse code (radiotelegraphy), which is considered the oldest maritime communication system; Medium Frequency (MF), Intermediate Frequency (IF), and High Frequency (HF)-based radio communications; as well as the international Very-High Frequency (VHF) and marine VHF communications, which emerged in the 1960s and 1990s, respectively [13].

While traditional maritime communication systems have served the industry well for many years, the limitations of radio waves have made it increasingly necessary to seek more advanced and efficient means of communication. For example, weather conditions and obstacles can affect radio wave communications, resulting in potential communication failures. Radio wave communications also have limited bandwidth, resulting in slower transmission speeds. In addition, the reliance of traditional maritime communication systems on manual methods of inputting, processing, and transmitting information can be time-consuming and error-prone. On the other hand, the integration of the internet into modern digital communications enhances real-time information sharing, collaboration and accuracy, which is essential in the fast-paced and dynamic maritime industry. With the development of Third Generation of Wireless Mobile Telecommunications Technology (3G) technology and its upgrades to Fourth Generation of Cellular Communications Standards (4G) and now Fifth Generation of Cellular Mobile Communications (5G), as well as the use of satellite communications, the internet has become more accessible and reliable in both inshore and offshore situations at sea.

3G is an upgrade of the Second Generation of Wireless Mobile Telecommunications Technology (2G), Second and a Half Generation of Wireless Mobile Telecommunications Technology (2.5G), General Packet Radio Service (GPRS), and Enhanced Data Rates for GSM Evolution (EDGE) technologies and is first researched in 1992 and first pre-commercialized in Japan in 1998. The 3G technology was initially able to increase the mobile communication rate of mobile phones to 144 kbps, and this figure was increased to several megabits per second in subsequent upgrades. 3G technology was gradually replaced around the 2010s by 4G, which allows mobile devices to access the internet at rates of up to 100 Mbps as long as they are in range, and has reached a 58% share of the global mobile telecommunications technology market by 2021 [14]. Today, Fifth Generation of Cellular Mobile Communications (5G) is replacing 4G as the new telecommunications standard. In addition, the use of technologies such as Wi-Fi extenders can also solve the problem of mobile internet access at sea in some near-shore navigation situations.

When sailing in distant seas, ships can access the internet via satellite. There is a decades-long history of using satellites for long-distance communications at sea, with Inmarsat founded in 1979 to ameliorate maritime telecommunications. With the first lauch of High Throughput Satellite (HTS) around 2004, people were able to access high-speed internet services from space for the first time. With the development of satellite communications, it is now possible to achieve high speed, low latency satellite internet access with up to 100 Mbps downlinks using the $K_u$-band[1] low earth orbiting satellites [15].

Table 1.1 lists some of the common technologies currently used for maritime digital communications as summarized by the International Association of Marine Aids to Navigation and Lighthouse Authorities (IALA) [16]. These technologies allow the exchange of maritime information in digital form.

**Table 1.1:** Digital communication technologies.

| Communication Technology | Data Rate | Infrastructure | Coverage | Transmission | Objective |
|---|---|---|---|---|---|
| NAVDAT | 12 ~ 18 kbps | NAVTEX | 250 / 300 NM | Broadcast | Maritime |
| VDES VDE | 307 kbps | VHF Data Link | 15 ~ 65 NM (w/o satellite) | Addressed / Broadcast | Maritime |
| VDES ASM | 19.2 kbps | VHF Data Link | 15 ~ 65 NM | Addressed / Broadcast | Maritime |
| Wi-Fi (IEEE 802.11ac) | 1,300 kbps | Routers / Access Points | 50 m | Addressed | Public |
| Digital VHF | 9.6 ~ 19.2 kbps | Base Station / Mobile Radios | 15 ~ 65 NM | Addressed | Maritime |
| Digital HF | 19.2 kbps | Base Station / Mobile Radios | Global | Addressed | Maritime |
| 4G (*incl.* LTE) | 600 Mbps | 4G Base Stations | 5 ~ 30 km | Addressed | Public |
| 5G | 1,200 Mbps | 5G Base Stations | 5 ~ 30 km | Addressed | Public |
| Inmarsat C | 600 bps | Satellite Service | Global, Spot Beams | Addressed / Broadcast | Maritime |
| Inmarsat GX | 50 Mbps | Satellite ($K_a$ Band) | Global, Spot Beams | Addressed / Broadcast | Cross Industry |
| Iridium | ~ 134 kbps | Satellite (L Band) | Global (constellation size) | Addressed / Broadcast | Cross Industry (Iridium Pilot Maritime) |

[1] The $K_u$ band is in the microwave range of frequencies from 12 to 18 GHz.

When accessing data through the internet (or similar computer networks), devices communicate over Internet Protocol (IP), the network layer communications protocol in the internet protocol suite[2]. There are many reasons why people choose IP communication, such as it is more mature, reliable[3], *etc.* With the increasing use of IP communications, there is an opportunity to exchange the following types of maritime information, which are currently sent and received using a variety of very different traditional means, in a more uniform and standardized manner in the future. For example, IALA has proposed their next-generation VHF Data Exchange System (VDES) that primarily uses VHF Data Link (VDL) for data exchange. In introducing VDES, IALA uses IP and VDL as parallel communication methods that serve both ship-to-shore and shore-to-ship communication from different aspects, as shown in Figure 1.1.

**Navigational Information (from the vessel)** This navigational information is transmitted from the ship, primarily via VHF, and includes the ship's identification, location, course, speed, destination, and other navigational data. The shipborne AIS equipment is currently responsible for sending and receiving this information [18]. The data sources are on-board electronic navigation sensors or manual input from the ship's officers.

**Navigational Information (from the administration)** This kind of navigational information is sent by authorities or maritime administrations, including various navigational warnings, notices, *etc.* Currently, these messages are mainly distributed through VHF, Navigational Telex (NAVTEX), *etc.*, and sometimes also on the web [19].

**Meteorological Information** Meteorological information includes weather forecasts and observational data, specifically temperature, visibility, weather forecasts (in text or weather chart format), *etc.* For ships, forecast information is needed from shore to guide navigation [20]. For shore, real-time observations from ships may be needed. Currently, meteorological information is mainly transmitted via NAVTEX, radiofacsimile (Weatherfax), Inmarsat, *etc.*

**Hydrographic Information** Hydrographic information includes information on water depth, navigational hazards such as wrecks and rocks, *etc.* Currently, hydrographic information is distributed mainly through the publication of nautical charts, notices to mariners, *etc.*, and sometimes also through NAVTEX, Inmarsat, *etc.* Under International Convention for the Safety of Life at Sea (SOLAS), all ships must have *adequate and up-to-date charts* to assist navigation [21].

**Reporting** The ship reporting system is adopted by many countries and regions around the world to ensure the safety of navigation to a greater extent. The types of ship reports include voyage plans, position reports, deviation reports, final reports, dangerous goods reports, harmful substances reports, marine pollutants reports, *etc.* [22]. Ship reporting is mainly done via VHF, Inmarsat, or Long-Range Identification and Tracking (LRIT).

**Emergency Message** In order to maximize the safety of human life at sea, International Maritime Organization (IMO) established Global Maritime Distress and Safety System (GMDSS) in the 1990s. In case of an emergency, distress signals and messages can be sent through various systems of GMDSS, including Emergency Position-Indicating Radio Beacon (EPIRB), NAVTEX, satellite, HF, Search and Rescue Transponde (SART), Digital Selective Calling (DSC), *etc.* [23].

[2] It is beyond the scope of this dissertation to discuss the internet protocol suite, seven-layer Open Systems Interconnection (OSI) model, and related concepts; please refer to Reference [17].

[3] In fact, IP by itself does not provide reliable communication; Transmission Control Protocol (TCP) and IP, together, provide a reliable service.

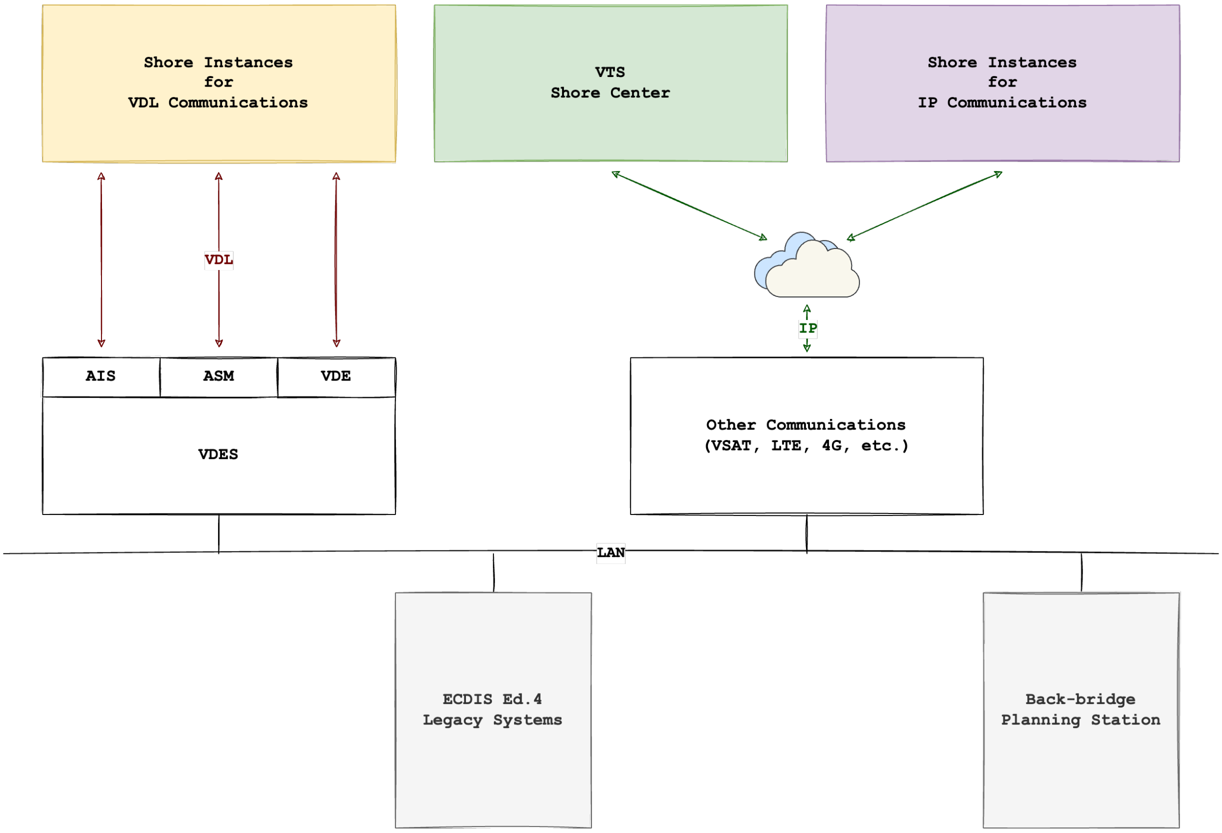


**Figure 1.1:** Concept for VDL and IP communications [16].

In addition, IP communication can also perform the function of general communication, such as liaison with shipping companies and agents ashore, *etc.*, and the consumers of maritime information are much broader than ship officers — for example, maritime pilots, students, researchers, ship masters, administrations, *etc.* [24–28], are all important users of maritime information.

However, despite the rapid development of maritime information and communication technology, the current state of maritime information management is still characterized by fragmented and decentralized systems with multiple sources of information and limited coordination between them [29]. The lack of a centralized platform for the collection and distribution of maritime information leads to inefficiencies and difficulties in accessing timely and accurate information, thus affecting the safety and efficiency of maritime operations. Specifically, some of the challenges the maritime industry faces in managing maritime information include:

**Data Integration** The growing number of electronic navigation sensors, ship management systems, and other sources of maritime data has led to an explosion of information. However, these sources often use different data formats, protocols, and data structures, making it difficult to integrate and use the data effectively. To address this challenge, a unified approach to data integration is needed, including the development of data standards and the use of common data protocols. This will enable the effective exchange and use of information, resulting in more efficient and safer operations.

**Timeliness** Maritime operations depend on timely, accurate, and relevant information. However, the collection and dissemination of real-time information in the maritime sector can be challenging due to a number of factors. For example, the vast geographic scope of the maritime sector makes it difficult to transmit and disseminate data in a timely manner. The solution to this challenge is a robust and efficient information system that can collect, process, and distribute maritime information in a timely manner. However, this may require investment in technology and infrastructure, as well as collaboration among stakeholders to ensure that information is shared in a standard format and that privacy and security are maintained.

**Information Security** As the maritime industry becomes more digitized, the risk of cyber-attacks, data breaches, and unauthorized access to sensitive information increases. To address these security challenges, the industry may need to implement comprehensive security measures such as firewalls, encryption, and multi-factor authentication. In addition, relevant stakeholders, such as governments and technology companies, must work together to establish security standards and protocols to ensure the secure exchange of maritime information.

**Data Quality** Several factors can affect the quality of maritime information, including errors in data collection, measurement, and processing, as well as inaccurate or outdated data sources. The sheer volume of data generated from multiple sources can also make it difficult to identify and correct errors in a timely manner. To overcome this challenge, maritime organizations must invest in robust data quality management processes, including data validation, standardization, and quality monitoring. This also requires collaboration between all parties involved in the data collection and processing cycle.

The current challenges in maritime information collection, management, and application highlight the need for a comprehensive solution in the design, development, and deployment of modern maritime information support systems. Based on these perspectives, this dissertation endeavors to provide a comprehensive approach to the design, development, and deployment of modern maritime information support systems, which takes into consideration some of the above challenges faced by the industry. In doing so, it aims to contribute to ongoing efforts to address these challenges and improve overall maritime information support.

## 1.2 The Main Ideas

As mentioned in Section 1.1, this research is motivated by the challenges and limitations faced by the maritime industry in collecting, managing, and applying maritime information. The overall objective of this dissertation is to propose a general solution for the design, development and deployment of modern maritime information support systems. In general, the system should be flexible, allowing for easy changes and modifications; scalable, ensuring that it continues to perform effectively as the system scales; and efficient, providing the right information at the right time.

Specifically, in terms of design, this dissertation seeks to achieve the following objectives:

- Identify an optimal system architecture that can effectively support the needs of a modern maritime information support system;
- Find suitable communication means to be used for both one-way (unidirectional) and two-way (bidirectional) data transfer at the IP communication level;
- Design and define clear communication interfaces between microservices as well as externally facing Application Programming Interfaces (APIs) to ensure seamless data transfer;
- Design tailored solutions and corresponding user-friendly frontend applications to address some practical problems of maritime information communication in different scenarios and with varying characteristics.

In terms of development, this dissertation aims to:

- Select the optimal set of programming languages, tools, and frameworks to efficiently develop the backend services and frontend applications of the system;
- Implement the design for the backend services and frontend applications with the selected programming languages, tools, and frameworks, to achieve the best possible performance and functionality.

In terms of deployment, this dissertation aims to:

- Highlight the benefits of the system's architecture in terms of flexibility, scalability, and other key features during the deployment phase;
- Provide a lightweight and streamlined deployment solution for specific, practical application scenarios that require simplicity;
- Demonstrate the ability to fully and efficiently deploy the system in more complex and demanding application scenarios, including through cloud deployment.

In addition to the above objectives, this dissertation also focuses on addressing practical issues in specific application scenarios, including:

- Offer a reliable and cost-effective solution for the exchange of navigational data for ships that are not mandated by the IMO to be equipped with AIS;
- Provide Three-Dimensional (3D) visualization of navigational data for Maritime Education and Training (MET), as well as lookout assistance;
- Provide a comprehensive range of data, including real-time and historical AIS data, vessel trajectory, statistical data, and other relevant information to maritime information consumers on demand;
- Design and implement effective communication services to provide information subscriptions, active reporting, and automated reporting capabilities, catering to the diverse needs of various stakeholders.

Due to time and resource limitations, this dissertation will focus primarily on the use of AIS data as a representative sample within the broader scope of maritime information. The proposed solutions and design considerations outlined in this dissertation focus on AIS or similar data because it provides both dynamic and static, real-time and historical information, making it a comprehensive representation. However, it is important to note that the proposed solutions are not limited to AIS data and can be adapted to meet the specific requirements of other data types and practical applications.

## 1.3 Outline

This dissertation is comprised of three main parts. The first part, Modern Software Architecture and Web Applications, covers Chapters 2 to 4 and delves into the software architecture of the system. The second part, Modern Communication Framework and Cross-Platform Applications, encompasses Chapters 5 and 6 and focuses on the communication between the system's services. Finally, the third part, Modern Deployment Approach and Microservices Containerization, which includes Chapters 7 and 8, examines the modern deployment approach for the system.

At the conclusion of the first part, a basic backend system for the provision of maritime information will be established along with the corresponding frontend applications. The second part will focus on improving inter-service communication through the use of optimized communication protocols and frameworks, as well as incorporating additional functionality through innovative communication methods. Finally, in the third part, the system will be deployed in various production environments to move beyond the laboratory phase and become a practical solution. The structure of this dissertation is outlined as follows:

***Chapter 1* Introduction** This chapter provides a comprehensive overview of the dissertation, including its background, motivation, objectives, and outline.

Modern Software Architecture and Web-Based Applications

***Chapter 2* Designing Microservices** This chapter delves into one of the main focuses of this dissertation, the microservices architecture, and provides an in-depth examination of the design of a system that provides AIS information, primarily from a backend perspective, using the microservices architecture. The chapter covers topics such as the system structure, RESTful APIs, request and response messages, *etc.*

***Chapter 3* Developing Microservices** This chapter covers the process of using AIS data as the primary source for the system, and explains the method used to parse the raw AIS messages. It also details the algorithms for extracting continuous ship trajectories from the discrete AIS data and compressing these trajectories for storage. The chapter also includes information on selecting a NoSQL database to store the processed AIS and trajectory data.

*Chapter 4* **Leveraging Microservices** This chapter focuses on developing both web applications and desktop applications based on web technologies. To meet the needs of different application scenarios such as ship navigation, meteorology, and traffic management, three applications are developed using modern web technologies and frameworks. These applications communicate with the backend services using the RESTful APIs developed in the previous chapter.

### Modern Communication Framework and Cross-Platform Applications

*Chapter 5* **Defining gRPC Services** This chapter explores the use of gRPC, a cutting-edge communication framework based on the HTTP/2 protocol. The chapter defines the communication service interface and the relevant messages, taking into account the different communication patterns of gRPC and the specific data requirements of different application scenarios. In addition, this chapter presents the design approach and code samples in the protocol buffers format.

*Chapter 6* **Implementing gRPC Services** This chapter implements the server and client sides of the gRPC services, using the design presented in the previous chapter. For the server side, the Go programming language was chosen because of its advantages, which will be discussed in this chapter. On the client side, a cross-platform application was developed, which offers distinct advantages over previous web technology-based applications. This chapter also introduces a new application scenario for cross-platform applications in the maritime industry and proposes a solution to a practical problem.

### Modern Deployment Approach and Microservices Containerization

*Chapter 7* **Deploying Microservices** This chapter delves into the packaging of microservices that were designed and developed in the previous chapters using modern containerization technologies. Taking advantage of the benefits of the microservices architecture, such as increased flexibility, reduced coupling, enhanced reliability, and improved scalability, the microservices are deployed through the Docker platform and the Kubernetes container orchestration system to accommodate various application scenarios at different scales.

*Chapter 8* **Testing Microservices** This chapter focuses on evaluating the performance of the previously designed, developed, and deployed microservices. It compares the advantages and disadvantages of the backend API implementations discussed in the dissertation with respect to request and response payload, server throughput, latency, resource consumption, and other key metrics. The discussions are based on the results of the experiments conducted.

*Chapter 9* **Conclusions and Outlook** This chapter summarizes the main findings of this dissertation, provides valuable insights and reflections on the research conducted, and highlights future research directions.

# Modern Software Architecture and Web Applications

# Designing Microservices 2

*Monolithic architecture* has been the conventional way of designing systems. However, this approach is known to have several shortcomings such as lack of flexibility and limited scalability, particularly in larger projects. As a result, *microservices architecture* has emerged as a promising solution to address these challenges and provide a more adaptable and scalable system design.

## 2.1 A Glimpse of Microservices

Microservices architecture is a modular approach to software design that differs from traditional monolithic architecture. In a monolithic architecture, all business logic is encapsulated together, leading to the deployment of the software system being deployed as a single unit [30]. In contrast, a microservices architecture allows the system to be divided into smaller, independent units that can be developed, deployed, and scaled individually.

Figure 2.1 provides a visual representation of a simple monolithic architecture, where all components are integrated and packaged as a monolithic *deployment bundle*. Communication within the system is facilitated through method calls [31, 32].

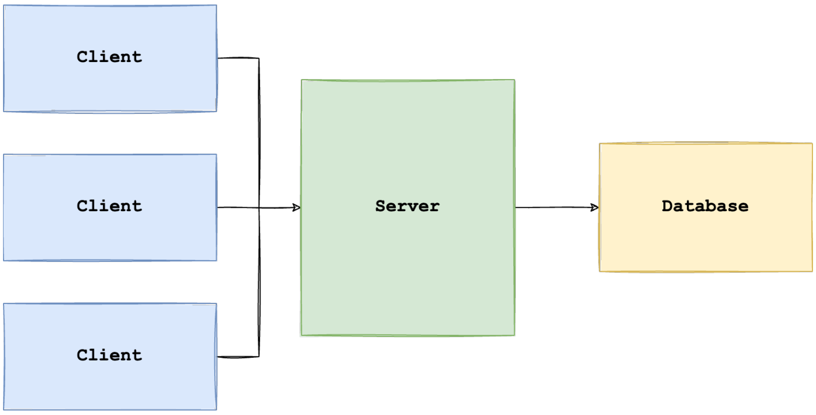


**Figure 2.1:** Monolithic architecture.

The performance of a single computing component is limited. To solve the problem of high concurrency and high availability in the architecture, redundancy is introduced. Deploying monoliths on multiple server components with load balancing, rather than on a single component, rationally distributes a large number of jobs across multiple operating units for execution, and can therefore improve the reliability of the overall system. Figure 2.2 illustrates a system with monolithic architecture and load balancing introduced.

Although software systems with monolithic architectures can also follow the modular mindset, meaning that each business logic can be structured, developed, and tested separately inside the monoliths, they must be deployed simultaneously on the same infrastructure and share the same

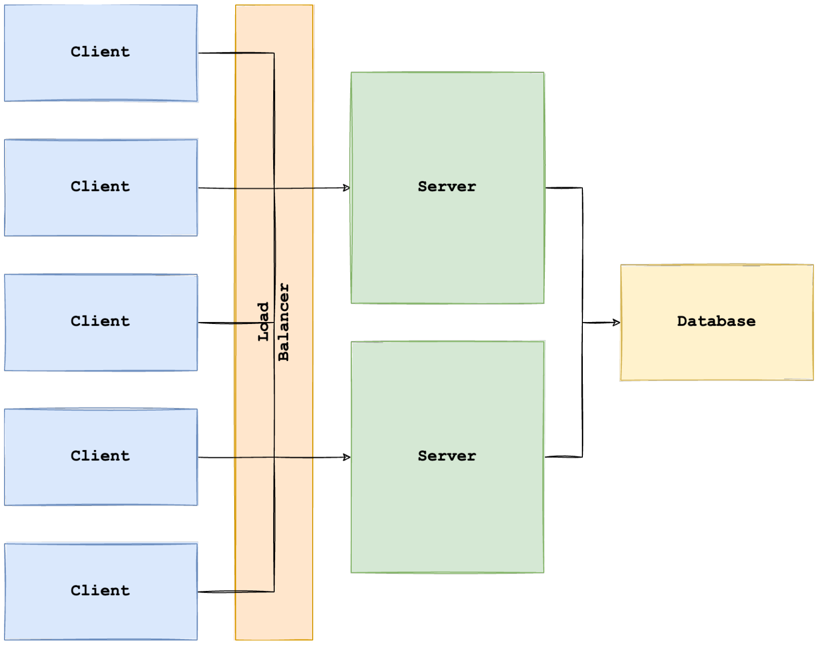


**Figure 2.2:** Monolithic architecture with load balancing.

resources, such as Central Processing Unit (CPU), memory, and storage. As a result, they suffer from maintainability and scalability issues [30, 33].

The advent of microservices architecture has alleviated these problems. A system based on microservices architecture is a collection of distinct, small, well-grained, single-purpose, and independently deployed services, typically organized by functionality. Microservices communicate over the network using Application Programming Interfaces (APIs) or messages [30].

Figure 2.3 gives an example of the microservices architecture. In this example, the client applications communicate only with the external-facing microservices. Once their communication interfaces are well designed, each microservice can be implemented, tested, and deployed independently. In this case, the database is deployed separately and is not exposed to other microservices, but only to the responsible one.

Microservices architecture has gained widespread popularity due to its significant advantages in technology heterogeneity, interchangeability, maintainability, scalability, and availability.

Following the idea of modularization, well-grained microservices can be designed, developed, and deployed independently, giving developers more freedom of choice in terms of technological implementation, including code base, platform, and database, depending on the need. Their highly modular nature makes it easy to replace a single component with even a different technology, as long as the same interface is maintained. Maintenance is also easier. Furthermore, unlike monoliths, which have to scale everything together when the performance bottleneck is reached, microservices can be scaled independently as needed (*e.g.* only the intensely requested ones). In addition, the failure of a microservice does not usually cascade. The failure may not affect other parts of the remaining system for the other functions, which also helps in troubleshooting. [30, 34]

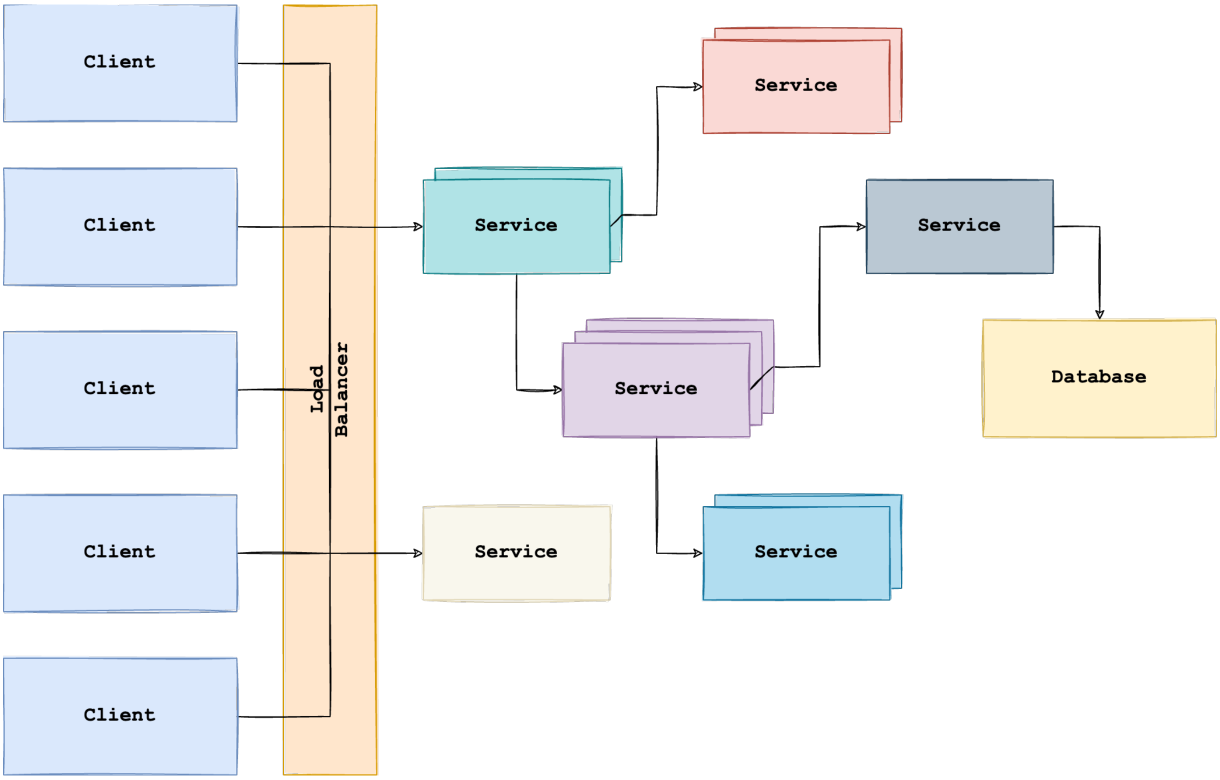


**Figure 2.3:** Microservices architecture.

## 2.2 Services in Pieces

When designing a modern maritime information support system, it's important to consider a microservices architecture. This approach granulates the system into functional pieces, which can improve scalability, flexibility, reliability, and maintainability. For example, in terms of scalability, if there's a sudden increase in demand for the exchange of navigational information, only the relevant microservice needs to be scaled up without affecting other services. A shipping company may need a service for tracking cargo, while a port authority may need a service to analyze the ship's historical trajectory, highlighting the importance of flexibility. Additionally, if one service, such as the weather information service, fails, it will not affect other services, such as the emergency notification service, thereby ensuring reliability. Moreover, microservices can be deployed across multiple servers, improving system availability and reliability. The microservices architecture also makes it easier to maintain and update the system without having to tear down the entire system to redeploy it.

This concept can be demonstrated through an example of a system that provides navigational information using AIS data, as illustrated in Figure 2.4. The diagram includes several separate microservices that work together to offer the necessary functionality. The individual microservices and components are listed as follows:

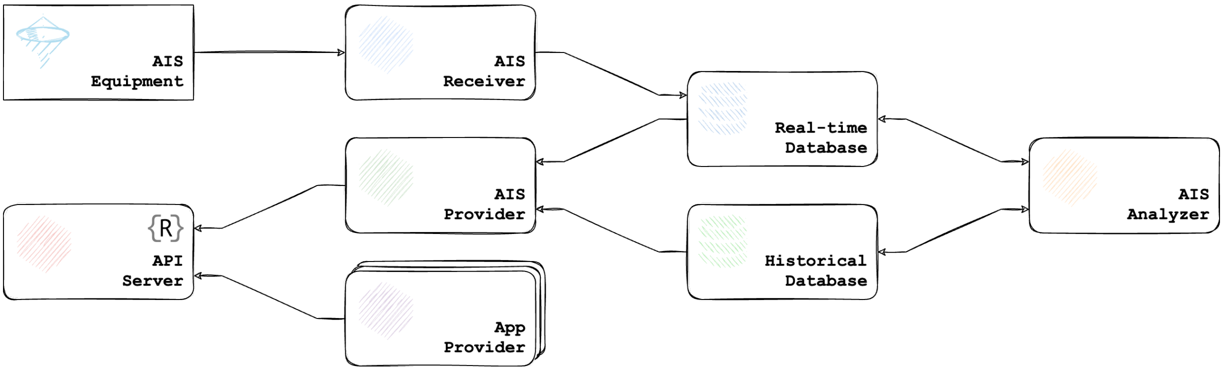


**Figure 2.4:** System components.

**AIS Equipment** The equipment installed on board a vessel that receives AIS messages broadcast by other AIS equipment via VHF and forwards them to the AIS Receiver microservice over the ship's Local Area Networks (LAN).

**AIS Receiver** A microservice responsible for receiving and parsing incoming AIS messages in real-time. The AIS Receiver parses the incoming messages from the AIS Equipment and stores the parsed data in the Real-Time Database.

**Real-Time Database** A database that stores real-time AIS data received from the AIS Receiver. The data is set with a Time To Live (TTL) value and is periodically purged of obsolete data based on the modified time and TTL to ensure timeliness.

**AIS Analyzer** A microservice responsible for analyzing raw AIS data for tasks such as statistics and trajectory extraction. The AIS Analyzer connects to both the Real-Time and Historical Databases and is responsible for periodic or full backups of real-time data to the Historical Database for long-term preservation.

**Historical Database** A database that stores historical AIS data from the Real-Time Database. The Historical Database is populated by the AIS Analyzer and provides access to historical data for analysis and other purposes.

**AIS Provider** A microservice that provides AIS-related data to external users. If the requested data is not available directly from the database, the AIS Provider forwards the request to the appropriate microservice, such as the AIS Analyzer.

**App Provider** A microservice that delivers the frontend application (User Interface (UI)) for users to interact with. Users access the web application provided by the App Provider to access the functionality of the system.

**API Server** The API entry point that sits between the user (frontend application) and other microservices. The API Server allows users to access the various microservices through a single interface, providing a seamless and consistent user experience.

## 2.3 Inter-Service Communication

Traditional monolithic systems integrate their various components through simple function calls, enabling them to work together. In contrast, in a microservices architecture, each component, or microservice, is relatively independent. This independence requires careful design of the communication mechanisms to enable effective collaboration between microservices.

There are two primary types of communication in a microservices architecture: synchronous and asynchronous[35]. In synchronous communication, one microservice sends a request and waits for a response from another microservice. This approach requires both microservices to maintain the connection until the response is received. In asynchronous communication, one microservice sends messages and finishes immediately without waiting for a response.

The choice between synchronous and asynchronous communication styles depends on the specific application scenario. For example, if the business logic of one microservice requires a response from another microservice before proceeding, synchronous communication is appropriate.

### Asynchronous Communication

Building on the previous discussion and the system design presented in Section 2.2, the system is designed to be response-insensitive when receiving AIS messages, meaning that it only needs to receive the message from the shipborne AIS equipment and initiate the parsing process without the need to return any result to the AIS equipment. To achieve this, the system utilizes asynchronous communication between the AIS equipment and the AIS Receiver microservice, as opposed to synchronous communication which would require both microservices to maintain a connection until a response is received. An example of the asynchronous communication messages used in the system is provided in Message 2.1.

**Message 2.1: Asynchronous AIS message.**

```
!AIVDM,1,1,,A,152lbV`000ad<<`CoB=UbTRJ0Dg:,0*7C
```

Further details on the interpretation of AIS messages are presented in Section 3.1.

### Synchronous Communication

In most cases, the system needs to be able to respond to user requests in a timely manner. For instance, when users request dynamic information about the ships in their vicinity, they expect the system to respond as quickly as possible. To achieve this, the system is designed to use synchronous communication in these scenarios.

One commonly used architectural design for synchronous communication is Representational State Transfer (REST), which creates distributed systems based on hypermedia [36]. In the REST model, resources are referred to as objects and services, and when the application accesses a resource using a Uniform Resource Identifier (URI), a representation of the resource is returned. Although Hypertext Transfer Protocol (HTTP) is the most commonly used implementation protocol for REST, it is protocol agnostic [35].

In REST, the Uniform Resource Locator (URL) of a resource serves as its identifier, and the *HTTP verbs* `GET`, `PUT`, `DELETE`, `POST`, *etc.* can be used to perform standard operations on the resource. The design of the REST architecture is based on synchronous communication.

Table 2.1 provides a brief overview of the request paths, parameters, and return values for some of the APIs. For examples of actual requests and responses for some of the APIs described in the table, see Requests 2.1 through 2.3 and Responses 2.1 through 2.3. For a more detailed description of utilizing the APIs, see Chapter 4.

**Request 2.1: Example requests to the `/dynamics` endpoint.**

```
GET https://{{baseUrl}}/api/v1/dynamics
Accept: application/json
```

**Table 2.1:** Design of some APIs.

| Endpoint URI | HTTP Method | Required Parameters | Optional Parameters | Description |
|---|---|---|---|---|
| `/dynamics` | `GET` | - | `populate`, `mmsi` | Query dynamic AIS data in the real-time database. |
| `/dynamics/near` | `GET` | `mmsi`, `range` | `populate` | Query dynamic AIS data near the given ship in the real-time database. |
| `/statics` | `GET` | - | `mmsi`, `name` | Query static AIS data in the real-time database. |
| `/statics/near` | `GET` | `mmsi`, `range` | - | Query static AIS data near the given ship in the real-time database. |
| `/history/dynamics` | `GET` | `timestamp` | `max_lat`, `max_lon`, `min_lat`, `min_lon`, `populate`, `mmsi` | Query dynamic AIS data at a given timestamp in the historical database. |
| `/history/statics` | `GET` | `timestamp` | `max_lat`, `max_lon`, `min_lat`, `min_lon`, `mmsi` | Query static AIS data at a given timestamp in the historical database. |
| `/trajectories` | `GET` | `start`, `end` | `max_lat`, `max_lon`, `min_lat`, `min_lon`, `populate`, `mmsi` | Query the trajectory data of a ship with the given area in the given time period. |
| `/dynamics` | `PUT` | N/A | N/A | Report real-time dynamic AIS data in the request body. |
| `/statics` | `PUT` | N/A | N/A | Report real-time static AIS data in the request body. |

**Response 2.1: Example response from the `/dynamics` endpoint.**

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "found": 311,
    "data": [
        {
            "mID": 1,
            "uID": 338504346,
            "tSt": 1610713800000,
            "nSt": 8,
            "SOG": 0.0,
            "COG": 145.0,
            "HDG": 145,
            "ROT": 0,
            "lat": 34.714544,
            "lon": 135.482700
        }, ...
    ]
}
```

**Request 2.2: Example request to the `/history` endpoint.**

```
GET https://{{baseUrl}}/api/v1/history/dynamics?
timestamp=1605870000000&populate=true
Accept: application/json
```

**Response 2.2: Example response from the `/history` endpoint.**

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "found": 311,
    "data": [
        {
            "mID": 3,
            "uID": 235050802,
            "tSt": 1605870000000,
            "nSt": 5,
            "SOG": 0,
            "COG": 309,
            "HDG": 40,
            "ROT": 0,
            "lat": 34.6495,
            "lon": 135.40063333333333
            "static": {
                "mID": 5,
                "uID": 235050802,
                "tSt": 1589401554362,
                "imo": 9384875,
                "cSg": "MAQJ",
                "vNm": "TOKYO TOWER",
                "typ": 70,
                "dim": [ 132, 40, 17, 10 ],
                "dft": 9.3,
                "dst": "JPOSK",
                "ETA": "2020-11-20T07:00+09"
            },
        }, ...
    ]
}
```

**Request 2.3: Example request to the `/trajectories` endpoint.**

```
GET https://{{baseUrl}}/api/v1/trajectories?
mmsi=431300065&start=1607583396082&end=1609426799999
Accept: application/json
```

**Response 2.3: Example response from the `/trajectories` endpoint.**

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "found": 38,
    "data": [
        {
            "metadata": {
                "from": {
                    "type": "Point",
                    "coordinates": [ 135.293166, 34.71653 ]
                },
                "to": {
                    "type": "Point",
                    "coordinates": [ 135.292805, 34.71719 ]
                },
                "len": 891,
                "start": 1579150252772,
                "end": 1579157989309,
            },
            "static": {
                "uID": 431300065,
                "imo": 8716710,
                "cSg": "JJ3518",
                "vNm": "FUKAEMARU",
                "typ": 99,
                "dim": [ 26, 24, 3, 7 ],
                "dft": 3.5,
                "dst": ">JP UKB XX",
                "ETA": "2020-01-16T07:30+09"
            },
            "geometry": [
                [ 34.716536, 135.293166 ],
                [ 34.716475, 135.29315 ],
                [ 34.716415, 135.29312 ],
                [ 34.716366, 135.293095 ],
                [ 34.716316, 135.29307 ],
                [ 34.716265, 135.29304 ], ...
            ],
        }, ...
    ]
}
```

## 2.4 Summary

Chapter 2 focused on a central aspect of this dissertation, the design of microservices. Microservices architecture has been proposed as a system architecture solution for larger projects, as traditional systems designed with monolithic architecture have certain problems, such as limited flexibility and scalability. The chapter begins with a broad overview of microservices architecture and its importance in shaping the design of modern maritime information support systems. It then examines the various microservices that make up the system, outlining their individual functions and purposes. The third section of the chapter focused on the intricacies of inter-service communication, specifically the design of the backend RESTful APIs that facilitate communication between microservices. This chapter serves as the foundation for the rest of the development and implementation of the system.

# Developing Microservices 3

AIS is a critical data source for navigation systems, and this chapter provides an in-depth look at AIS as a data source for the system. It describes the technique for parsing encapsulated raw AIS messages and outlines algorithms for extracting continuous trajectories from discrete AIS dynamic data as well as compressing the extracted trajectories. Finally, a NoSQL database is selected to store the corresponding AIS and trajectory data.

## 3.1 Handling Incoming AIS Data

The AIS Receiver microservice is responsible for receiving messages from AIS devices. These messages are encapsulated, which means that they must be parsed before the information they contain can be accessed. Therefore, the AIS Receiver microservice is designed and developed to parse these encapsulated messages and extract the relevant navigational information from them.

### 3.1.1 Different Types of AIS Messages

The AIS system consists of 27 different message types, each with a specific function, as described in the *Recommendation R-REC-M.1371: Technical characteristics for an automatic identification system using time division multiple access in the VHF maritime mobile frequency band* and shown in Table 3.1 [37]. The messages transmitted by AIS devices can be categorized into three types: dynamic, static, and voyage-related. Dynamic messages provide real-time navigational information on a vessel's position, Speed Over Ground (SOG), and Course Over Ground (COG). Static messages provide static navigational information such as vessel name, call sign, and dimensions. Voyage-related messages contain navigational information manually entered by the ship officer for each voyage, such as destination and Estimated Time of Arrival (ETA). To process AIS data, the AIS Receiver microservice is designed to filter and parse eight relevant AIS message types: Identifiers (IDs) 1 to 3, 5, 18, 19, 24, and 27, depending on the system design.

**Table 3.1:** AIS messages.

| ID | Name | Description |
|---|---|---|
| 1 | Position report | Scheduled position report (Class-A shipborne mobile equipment) |
| 2 | Position report | Assigned scheduled position report (Class-A shipborne mobile equipment) |
| 3 | Position report | Special position report, response to interrogation (Class-A shipborne mobile equipment) |
| 4 | Base station report | Position, UTC, date and current slot number of base station |
| 5 | Static and voyage related data | Scheduled static and voyage related vessel data report; (Class-A shipborne mobile equipment) |
| 6 | Binary addressed message | Binary data for addressed communication |
| 7 | Binary acknowledgement | Acknowledgement of received addressed binary data |
| 8 | Binary broadcast message | Binary data for broadcast communication |
| 9 | Standard SAR aircraft position report | Position report for airborne stations involved in SAR operations, only |
| 10 | UTC/date inquiry | Request UTC and date |
| 11 | UTC/date response | Current UTC and date if available |
| 12 | Addressed safety related message | Safety related data for addressed communication |
| 13 | Safety related acknowledgement | Acknowledgement of received addressed safety related message |
| 14 | Safety related broadcast message | Safety related data for broadcast communication |
| 15 | Interrogation | Request for a specific message type (can result in multiple responses from one or several stations) |
| 16 | Assignment mode command | Assignment of a specific report behaviour by competent authority using a base station |
| 17 | DGNSS broadcast binary message | DGNSS corrections provided by a base station |
| 18 | Standard Class-B equipment position report | Position report & standard position report for Class-B shipborne mobile equipment to be used instead of Messages 1 to 3 |
| 19 | Extended Class-B equipment position report | No longer required; extended position report for Class-B shipborne mobile equipment; contains additional static information |
| 20 | Data link management message | Reserve slots for base station(s) |
| 21 | Aids-to-navigation report | Position and status report for aids-to-navigation |
| 22 | Channel management | Management of channels and transceiver modes by a base station |
| 23 | Group assignment command | Assignment of a specific report behaviour by competent authority using a base station to a specific group of mobiles |
| 24 | Static data report | Additional data assigned to an MMSI |
| 25 | Single slot binary message | Short unscheduled binary data transmission (broadcast or addressed) |
| 26 | Multiple slot binary message with communications state | Scheduled binary data transmission (broadcast or addressed) |
| 27 | Position report for long-range applications | Class-A and Class-B *SO* shipborne mobile equipment outside base station coverage |

Different AIS messages contain different information, as shown in Figure 3.1, but follow the exact encoding specification, namely NMEA 0183 data specification, created and maintained by the National Marine Electronics Association (NMEA), formatted as Message 3.1.

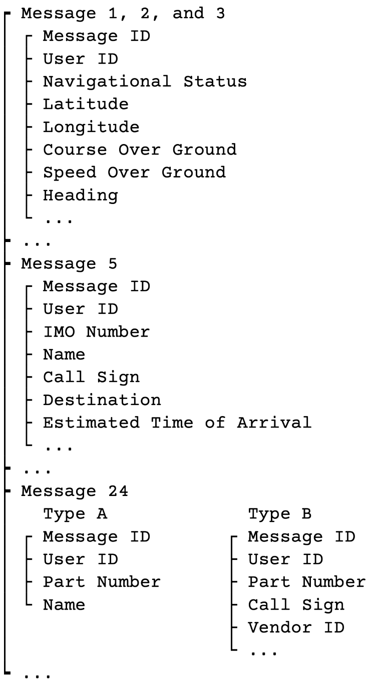


**Figure 3.1:** AIS message structure.

**Message 3.1: AIS message format.**

```
!AABBB,c,d,e,F,g-g,h*II<CR><LF>
```

Table 3.2 shows the reserved characters and their uses in the NMEA 0183 specification, while Table 3.3 explains the meanings of the remaining parts, represented by the letters `A` through `I`.

As an example of a typical AIS message, consider the partial message shown in Message 3.2, sent from a shipborne AIS device and encapsulated according to the NMEA 0183 specification:

**Message 3.2: AIS message example.**

```
!AIVDM,1,1,,A,152lbV`000ad<<`CoB=UbTRJ0Dg:,0*7C
```

**Table 3.2:** Reserved characters used in NMEA 0183 format messages.

| ASCII | Hex | Use |
|---|---|---|
| `<LF>` | 0x0a | Line feed, end delimiter |
| `<CR>` | 0x0d | Carriage return |
| `!` | 0x21 | Start of encapsulation sentence delimiter |
| `$` | 0x24 | Start delimiter |
| `*` | 0x2a | Checksum delimiter |
| `,` | 0x2c | Field delimiter |
| `\` | 0x5c | TAG block delimiter |
| `^` | 0x5e | Code delimiter for HEX representation of ISO/IEC 8859-1 (ASCII) characters |
| `~` | 0x7e | Reserved for further use |

**Table 3.3:** Meaning of each part in Message 3.2.

| Part | Meaning |
|---|---|
| `AA` | Talker identifier, `AI` for AIS |
| `BBB` | Sentence formatter, `VDO` for the ownship, and `VDM` for other vessels |
| `c` | Total number of sentences of the message |
| `d` | Sentence sequential number |
| `e` | Sequential message ID |
| `F` | AIS channel |
| `g-g` | Encapsulated data |
| `h` | Data end |
| `II` | Bitwise XOR checksum result of characters between `$`/`!` and `*` (not inclusive) |

### 3.1.2 Parsing AIS Messages

The received AIS messages are encapsulated, and must be parsed before being written to the database and used by other microservices. Parsing the messages involves reducing the American Standard Code for Information Interchange (ASCII) characters one by one to 6-bit binary code, according to the encoding rules in *Recommendation R-REC-M.1371*, and then grouping and converting them to decimal numbers to obtain the specific data values. The `g-g` field in Message 3.1 encodes dynamic, static, and voyage-related information, and needs to be parsed first, depending on the message type. In case of messages consisting of multiple sentences, such as messages of type 5, it is necessary to wait until all the sentences are received and then perform the parsing work by concatenating the `g-g` parts according to the sentence sequence number (part `d` in Message 3.1).

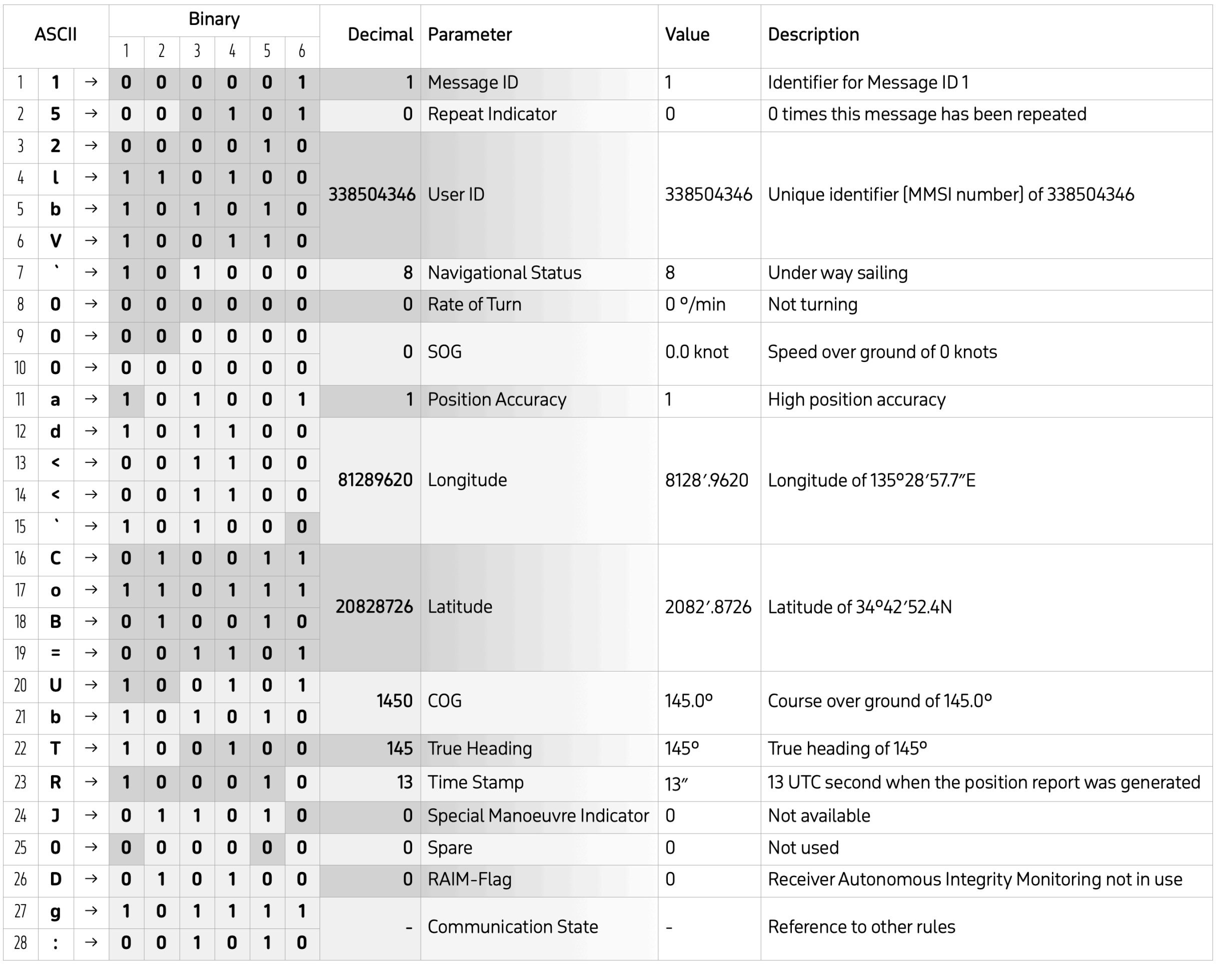
Figure 3.2 illustrates how dynamic AIS information in messages with IDs 1 to 3 is decapsulated from the `g-g` part, using Message 3.2 as an example. The `g-g` part consists of 28 ASCII characters, reduced to 168 bits and divided into 16 sections according to the specification. The groups of bits are shown along with the information they represent within the message. The method of grouping the bits varies for different message IDs and a general flow of the decoding is given in Algorithm 3.1, which also includes a checksum algorithm (Algorithm 3.2) to ensure the integrity of the received messages and the accuracy of the resulting data.

**Algorithm 3.1:** Parsing AIS Message

```
Name   : ParseAIS
Param  : AIS message msg
Param  : (optional) Previously stored partial bit message prevBitMsg
Return : Parsed AIS data doc
Return : Any error err during parsing process
doc, err ← $\phi$;
recvTime ← current time;
recvTime → doc;
if XORCheck (msg) then
    asciiStr ← extract g-g part from msg;
    bitMsg ← $\phi$;
    foreach char ∈ asciiStr do
        convert char to 6-bit binary number → bitMsg;
    get sentence number sNo and total sentences from msg;
    if sNo = 1 then
        parse message ID mID from bitMsg;
        mID → doc;
        if mID ∈ $\{1, 2, 3, 18, 19, 24\}$ then
            parse remaining items according to specification in
              Recommendation R-REC-M.1371 → doc;
        else if mID = 5 then
            store bitMsg for further parsing process, passing as
              prevBitMsg;
        else
            err ← Unsupported message type
    else if sNo = 2 then
        if prevBitMsg $\not\equiv \phi$ then
            bitMsg → prevBitMsg;
            parse remaining items according to specification in
              Recommendation R-REC-M.1371 → doc;
        else
            err ← Missing prefix message
else
    err ← Message integrity check failed
```

**Algorithm 3.2:** Checksum

```
Name   : XORCheck
Param  : AIS message msg
Return : Check result passed
passed ← False;
sum ← 0;
str, checksum ← Extracts the string str to be verified and the
  expected result checksum from msg;
foreach char ∈ str do
    ord ← get ASCII code of char;
    sum ← sum $\oplus$ ord;
sum ← Convert sum to hexadecimal;
if sum = checksum then
    passed ← True;
```



| ASCII | | | Binary | | | | | | Decimal | Parameter | Value | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 | 6 | | | | |
| 1 | **1** | → | **0** | **0** | **0** | **0** | **0** | **1** | 1 | Message ID | 1 | Identifier for Message ID 1 |
| 2 | **5** | → | **0** | **0** | **0** | **1** | **0** | **1** | 0 | Repeat Indicator | 0 | 0 times this message has been repeated |
| 3 | **2** | → | **0** | **0** | **0** | **0** | **1** | **0** | 338504346 | User ID | 338504346 | Unique identifier (MMSI number) of 338504346 |
| 4 | **l** | → | **1** | **1** | **0** | **1** | **0** | **0** | | | | |
| 5 | **b** | → | **1** | **0** | **1** | **0** | **1** | **0** | | | | |
| 6 | **V** | → | **1** | **0** | **0** | **1** | **1** | **0** | | | | |
| 7 | **`** | → | **1** | **0** | **1** | **0** | **0** | **0** | 8 | Navigational Status | 8 | Under way sailing |
| 8 | **0** | → | **0** | **0** | **0** | **0** | **0** | **0** | 0 | Rate of Turn | 0 °/min | Not turning |
| 9 | **0** | → | **0** | **0** | **0** | **0** | **0** | **0** | 0 | SOG | 0.0 knot | Speed over ground of 0 knots |
| 10 | **0** | → | **0** | **0** | **0** | **0** | **0** | **0** | | | | |
| 11 | **a** | → | **1** | **0** | **1** | **0** | **0** | **1** | 1 | Position Accuracy | 1 | High position accuracy |
| 12 | **d** | → | **1** | **0** | **1** | **1** | **0** | **0** | 81289620 | Longitude | 8128′.9620 | Longitude of 135°28′57.7″E |
| 13 | **<** | → | **0** | **0** | **1** | **1** | **0** | **0** | | | | |
| 14 | **<** | → | **0** | **0** | **1** | **1** | **0** | **0** | | | | |
| 15 | **`** | → | **1** | **0** | **1** | **0** | **0** | **0** | | | | |
| 16 | **C** | → | **0** | **1** | **0** | **0** | **1** | **1** | 20828726 | Latitude | 2082′.8726 | Latitude of 34°42′52.4N |
| 17 | **o** | → | **1** | **1** | **0** | **1** | **1** | **1** | | | | |
| 18 | **B** | → | **0** | **1** | **0** | **0** | **1** | **0** | | | | |
| 19 | **=** | → | **0** | **0** | **1** | **1** | **0** | **1** | | | | |
| 20 | **U** | → | **1** | **0** | **0** | **1** | **0** | **1** | 1450 | COG | 145.0° | Course over ground of 145.0° |
| 21 | **b** | → | **1** | **0** | **1** | **0** | **1** | **0** | | | | |
| 22 | **T** | → | **1** | **0** | **0** | **1** | **0** | **0** | 145 | True Heading | 145° | True heading of 145° |
| 23 | **R** | → | **1** | **0** | **0** | **0** | **1** | **0** | 13 | Time Stamp | 13″ | 13 UTC second when the position report was generated |
| 24 | **J** | → | **0** | **1** | **1** | **0** | **1** | **0** | 0 | Special Manoeuvre Indicator | 0 | Not available |
| 25 | **0** | → | **0** | **0** | **0** | **0** | **0** | **0** | 0 | Spare | 0 | Not used |
| 26 | **D** | → | **0** | **1** | **0** | **1** | **0** | **0** | 0 | RAIM-Flag | 0 | Receiver Autonomous Integrity Monitoring not in use |
| 27 | **g** | → | **1** | **0** | **1** | **1** | **1** | **1** | - | Communication State | - | Reference to other rules |
| 28 | **:** | → | **0** | **0** | **1** | **0** | **1** | **0** | | | | |

**Figure 3.2:** AIS message parse.

## 3.2 Preparing Trajectory Data

AIS data are discrete, *i. e.*, the dynamic information is represented as discrete points when plotted on a Two-Dimensional (2D) plane. To obtain the ship's trajectory data, additional processing of the AIS data is necessary.

### 3.2.1 Extracting Trajectories from AIS Data

The first step in preparing the trajectory data is to extract the continuous trajectories from the discrete dynamic AIS data. This involves segmenting the data according to certain rules to determine which data points belong to the same trajectory. Since AIS data is received and stored in a time series, the segmentation process is based on time intervals.

Algorithm 3.3 provides the steps for simultaneous trajectory recording when dynamic AIS information is received. This algorithm can also be applied to extract trajectories from historical AIS data.

**Algorithm 3.3:** Trajectory Recording / Extraction

```
Name  : TrajectoryRecord
Param : Received AIS dynamic data data
Param : (optional) Trajectory data being processed trajectory
if data is valid then
    if trajectory $\equiv \phi$ then
        data $\rightarrow$ trajectory;
        nextData $\leftarrow$ wait for / get next dynamic data;
        trajectory $\leftarrow$ TrajectoryRecord (nextData, trajectory);
    else
        lastTime $\leftarrow$ receive time of the last data in trajectory;
        thisTime $\leftarrow$ receive time of data;
        if thisTime $-$ lastTime $>$ threshold then
            Add metadata $\rightarrow$ trajectory;
            Do statistics $\rightarrow$ trajectory;
            if trajectory is valid then
                Save trajectory;
            trajectory $\leftarrow$ TrajectoryRecord (data, $\phi$);
        else
            data $\rightarrow$ trajectory;
            nextData $\leftarrow$ wait for / get next dynamic data;
            trajectory $\leftarrow$ TrajectoryRecord (nextData, trajectory);
else
    nextData $\leftarrow$ wait for / get next dynamic data;
    trajectory $\leftarrow$ TrajectoryRecord (nextData, trajectory);
```

### 3.2.2 Compressing Trajectory Data

AIS messages can be broadcast as often as once every 2 seconds, which can result in a considerable amount of data when tracking a vessel. The transmission of large amounts of data can cause problems related to transmission rate, stability, and data integrity. As a result, it is common practice to compress trajectory data to address these issues. Compression can be achieved using appropriate algorithms that still ensure the accuracy of the data.

Algorithms 3.4 and 3.5 provide a trajectory compression algorithm based on the Ramer-Douglas-Peucker algorithm, which uses the *divide-and-conquer* approach [38, 39]. The Ramer-Douglas-Peucker algorithm simplifies a polyline by recursively eliminating intermediate points that can be accurately approximated by the remaining points.

**Algorithm 3.4:** Trajectory Compression

```
Name  : TrajectoryCompression
Param : Trajectory data $trajectory$
Param : Tolerance $\epsilon$ from a compressed point to the trajectory line
Return: Compressed trajectory $compressed$
$indices, compressed \leftarrow \phi$;
$0 \rightarrow indices$;
CompressionStep ($trajectory, 0, |trajectory|, \epsilon, indices$);
$|trajectory| - 1 \rightarrow indices$;
foreach $data \in trajectory$ do
    if index of $data \in indices$ then
        $data \rightarrow compressed$
```

**Algorithm 3.5:** Trajectory Compression Step

```
Name  : CompressionStep
Param : Trajectory data $trajectory$
Param : Index of the first point $first$ to be compressed
Param : Index of the last point $last$ to be compressed
Param : Tolerance $\epsilon$ from a compressed point to the trajectory line
Param : Indices of the compressed trajectory points $indices$
$index \leftarrow \phi$;
$maxDistance \leftarrow \epsilon$;
foreach $idx$, $first < idx < last$ do
    $distance \leftarrow$ distance from the trajectory data point with index $idx$
      to the line segment from the point with index $first$ to the point
      with index $last$;
    if $distance > maxDistance$ then
        $index \leftarrow idx$;
        $maxDistance \leftarrow distance$;
if $maxDistance > \epsilon$ then
    if $index - first > 1$ then
        CompressionStep ($trajectory, first, index, \epsilon, indices$);
    $index \rightarrow indices$;
    if $last - index > 1$ then
        CompressionStep ($trajectory, index, last, \epsilon, indices$);
```

## 3.3 Integrating with Databases

Databases can be categorized into different types, such as relational databases, NoSQL databases, *etc.*, each with its own strengths and weaknesses. Relational databases use a table structure to store data and have been around for a long time, making them a mature technology. On the other hand, NoSQL databases support more flexible data structures, making them ideal for unstructured and complex data. In this dissertation, MongoDB, a popular NoSQL database, is chosen as the database solution due to its ability to handle large and complex datasets.

### 3.3.1 NoSQL and MongoDB

Traditional relational databases are designed to manage structured and relatively static data, with data sets stored in predefined tables, as shown in Figure 3.3. While suitable for many complex applications, relational databases struggle with unstructured data and are difficult to scale as the size and complexity of the data increases. In addition, the mismatch between the relational model and the data structure in memory, known as *impedance mismatch*, can also be a bottleneck for relational databases [40]. These challenges gave rise to NoSQL databases, which support more flexible data structures and can handle unstructured and complex data more effectively. MongoDB is a widely-used NoSQL database that can overcome many of the challenges of relational databases and is well suited to the needs of modern maritime information support systems.

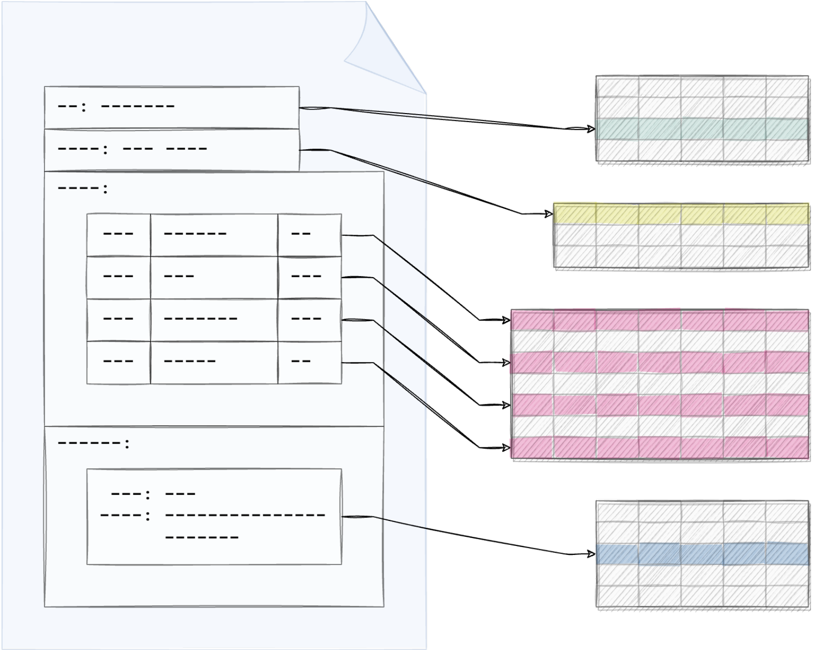


**Figure 3.3:** Relational database schema.

The original definition of NoSQL referred to database systems that could store unstructured data based on a multidimensional relational model, making it a better choice than traditional relational databases for managing rapidly growing and almost unlimited amounts of ship navigational data, such as AIS data. NoSQL databases can be classified into different types based on their characteristics and application scenarios, including key-value, graph, column, and document-oriented databases.

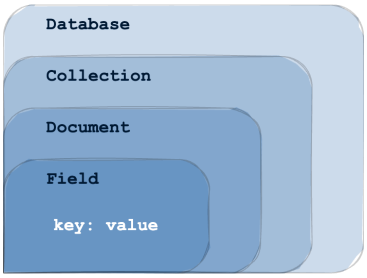


**Figure 3.4:** MongoDB database schema.

This dissertation selects the document-oriented MongoDB, which has several advantages over relational databases, including a flexible structure, high scalability, and the ability to handle high concurrency and large volumes of data. MongoDB uses an *expressive query language* that efficiently handles complex queries, despite not using Structured Query Language (SQL) like relational databases. Figure 3.4 illustrates the data storage structure of MongoDB.

### 3.3.2 Database Structure

```
┌ Real-time Database
│ ┌ Dynamic Data Collection
│ │   Type 1-3            Type 18
│ │ ┌ Message ID        ┌ Message ID
│ │ ├ User ID           ├ User ID
│ │ ├ Timestamp         ├ Timestamp
│ │ ├ Nav. Status       ├ SOG
│ │ ├ SOG               ├ COG
│ │ ├ COG               ├ HDG
│ │ ├ HDG               └ Geo. Data
│ │ ├ Rate of Turn
│ │ └ Geo. Data
│ ├ Static Data Collection
│ │   Type 5             Type 24
│ │ ┌ Message ID        ┌ Message ID
│ │ ├ User ID           ├ User ID
│ │ ├ Timestamp         ├ Timestamp
│ │ ├ IMO Number        ├ Call Sign
│ │ ├ Call Sign         ├ Ship Type
│ │ ├ Name              ├ Dimension
│ │ ├ Ship Type         └ Name
│ │ ├ Dimension
│ │ ├ Draft
│ │ ├ Destination
│ │ └ E.T.A.
│ ├ Trajectory Collection
│ │ ┌ Metadata
│ │ │ ┌ Length
│ │ │ ├ Start Time
│ │ │ ├ End Time
│ │ │ ├ Start Location
│ │ │ └ End Location
│ │ ├ Dynamic Data
│ │ │   ...
│ │ ├ Static Data
│ │ │   ...
│ │ ├ Statistics
│ │ │ ┌ Time Span
│ │ │ ├ Average SOG
│ │ │ ├ Maximum SOG
│ │ │ ├ Weighted COG
│ │ │ └ ...
│ │ └ ...
│ └ ...
├ History Database
│ ┌ Dynamic Data Collection
│ │   ...
│ ├ Static Data Collection
│ │   ...
│ ├ Trajectory Collection
│ │   ...
│ └ ...
└ ...
```

**Figure 3.5:** Database design.

The database structure is shown in Figure 3.5 and consists of three layers: the database layer, the collection layer, and the document layer, where each document contains several key-value pairs. To store data from AIS devices, real-time data are collected and stored in the real-time database with a predefined lifetime, or TTL, after which it is deleted by the database management system. For microservices to use historical data, real-time data is synchronized and backed up to the historical database, which retains the data indefinitely.

The specific description of the databases is as follows:

**Real-Time Database** The real-time database stores the real-time data in collections. The collections in the real-time database are designed with a TTL feature that allows the database management system to automatically delete data that exceeds the expiration time. This ensures that the database contains only the most current and relevant data.

**Historical Database** The historical database stores data that has been synchronized from the real-time database. It can be implemented in two ways: as a live incremental backup of the live database that contains all the information received, or as an incomplete backup that is synchronized at intervals to reduce the size of the database.

To store dynamic and static data separately, both the real-time and historical databases contain a collection of dynamic data and a collection of static data. This approach is taken due to the different characteristics of how often messages are sent, what information they contain, and how often they are used. Additionally, a special trajectory collection is used to store the trajectory data extracted in Section 3.2.

**Dynamic Information Collection** The dynamic data collection is responsible for storing parsed ship dynamic information, particularly from messages with IDs 1 to 3, 18, and 19.

**Static Information Collection** The static data collection stores parsed ship static information, particularly from messages with IDs 5, 24, and 19.

**Trajectory Collection** The trajectory collection, on the other hand, is used to store the extracted historical trajectories of the ships. The trajectory extraction process can be either online or offline, depending on the needs and the chosen algorithm.

## 3.4 Summary

Chapter 3 focused on the critical role of AIS data, the representative real-time navigational information, in modern maritime information support systems, and the development of microservices to manage this data effectively. The chapter covers various aspects of AIS data handling, including the identification of different types of AIS messages, the parsing process, and the algorithms for extracting and compressing trajectory data. The chapter also examines the integration of the system with MongoDB, a NoSQL database solution, highlighting its strengths in terms of flexibility and scalability. Overall, this chapter provides a comprehensive understanding of the techniques and tools required to efficiently handle AIS data in modern maritime information support systems.

# Leveraging Microservices 4

In the previous chapters, significant progress was made in designing and implementing the necessary APIs to retrieve data from the backend system. However, the development process is not yet complete, as this chapter focuses on creating the client side of the applications. To achieve this, the unique data requirements of different users need to be considered. Three different web applications will be designed and developed to meet these needs. The goal is to create intuitive applications that can effectively utilize maritime information, while also providing a seamless experience for the end user, ensuring that users can quickly and easily access the information they need. This requires careful consideration of various factors, such as the application's intended use cases and data processing requirements.

## 4.1 BlueNavi: Providing 2D Visualization

The integration of AIS data into the Electronic Chart Display and Information System (ECDIS) allows for the real-time display of surrounding vessel identification and navigational information on electronic charts, thus supporting safe navigation. However, the lack of a mandate for smaller vessels to carry AIS, and the high cost of ECDIS installation, limit access to this critical information. To overcome these challenges, *BlueNavi*, a frontend application designed to provide real-time AIS information over IP communications, has been developed. By offering a cost-effective solution, BlueNavi makes AIS information available to vessels that may not have access to ECDIS, thereby increasing the availability of this important data.

### 4.1.1 Background and Scenarios

Despite the SOLAS regulation requires all ships of 300 Gross Tonnage (GT) or more engaged in international voyages, ships of 500 GT or more not engaged in international voyages, and all passenger ships regardless of size to carry AIS since 2008 [21], many smaller vessels, such as recreational craft, fishing vessels, and vessels between 300 and 500 GT that are not engaged in international voyages, are still not covered by the regulation [18, 41]. The smaller size of these vessels makes it difficult for other ships to detect and identify them using passive systems such as radar, and they cannot communicate with other ships using AIS.

A modern maritime information support system with microservices architecture is designed and developed to address this problem. The system can be deployed on a range of devices, including regular computers and even single-board computers such as the Raspberry Pi[1]. The system uses Class-B AIS devices as the primary data source, which are more suitable for small vessels due to their lower cost and compliance with international standards. However, Class-B AIS can only receive and transmit

[1] A credit-card-sized single-board computer that is designed to promote computer science education and enable Do-It-Yourself (DIY) electronics projects. The specific discussion of system deployment is presented in Chapter 7.

encapsulated AIS messages but has no display capabilities[2]. Therefore, a user interface for providing 2D visualization is essential.

While ECDIS are commonly used in the industry to display AIS data, they can be too expensive and bulky for smaller vessels. Moreover, commercial systems are often highly closed, limiting the ability to customize features. To make AIS information more accessible, a modern frontend application, named BlueNavi, is designed and developed to provide a tangible visual form of the received navigational data. BlueNavi is based on web technologies and can either be accessed via web browsers or run natively as a desktop application.

### 4.1.2 Tooling and Framework

Since BlueNavi uses mainly web technologies, this section will first discuss what a web application is and the reasons for choosing web technologies, followed by the specific choice of development tools and frameworks.

#### Web Applications

A *web application* is a computer program that uses a web browser to render interfaces and interact with users. As shown in Figure 4.1, web applications are usually composed of server-side (backend) and client-side (frontend).

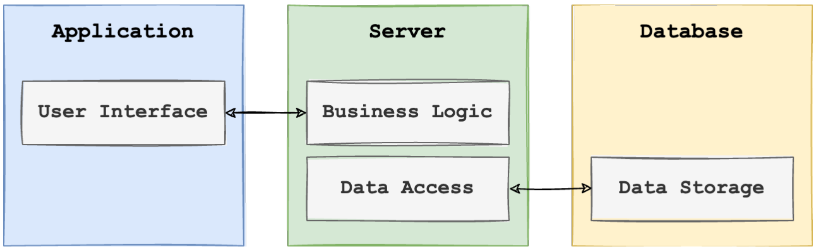


**Figure 4.1:** Web application schematic.

The backend is mainly responsible for the core data processing and business logic, while the frontend is mainly responsible for displaying data and interacting with users. As the name implies, the entire web application is based on web technologies, and the frontend is often delivered to users over the IP communication. Specifically, when requesting a web application, the user opens a web browser and sends a request to the server, which receives the request and sends the application's structure, usually handled by Hyper Text Markup Language (HTML) files; styles and layout, usually handled by Cascading Style Sheets (CSS) files; and execution logic and behavior, usually handled by JavaScript (JS) files; over the network to the client. The browser can then render the frontend and interact with the user. In the interaction process, if data requests or heavy data processing tasks are encountered, several more such request-response communications can be performed between the frontend and backends. In addition, unlike simple websites, web applications are more *interactive* (*i. e.*, functional).

The opposite of web applications are *native applications*. Unlike native applications, web applications require no installation and only a modern

[2] In fact, even the Class-A AIS device can only provide very limited display through its Minimum Keyboard Display (MKD) unit and not enough for actual ship handling and collision avoidance.

browser (*e. g.*, Safari, Chrome, *etc.*). Since modern browsers are a must for almost all device platforms (desktop, laptop, smartphone, tablet, *etc.*) and operating systems (Windows, macOS, iOS, Android, *etc.*) today, web applications have inherent cross-platform support compared to native applications.

### Desktop Applications using Web Technologies

Although web applications have the advantage of being accessible from any device with a browser and a network connection and do not need to be manually installed or updated, they have the drawbacks of limited offline functionality (web applications rely on a constant network connection to function), accessibility of native features (web applications cannot directly access native features of the operating system, such as the file system), compatibility (some browser compatibility issues may occur since it is not possible to predict the type and version of browser used by the user), *etc.* Most of these problems stem from the fact that users have to use the web application through a browser, which also affects the user experience.

In order to solve as many of these problems as possible, while retaining the benefits of web applications, new frameworks have been created, such as Electron.js. Using web technologies such as JavaScript, HTML, and CSS, people can create cross-platform desktop applications using the Electron.js framework. It is built on top of Node.js, which enables the server-side use of JavaScript, and Chromium, the open-source variant of Google Chrome, which serves as the rendering engine for the application's UI. This means that programmers can easily incorporate native features while still using the same codebase to produce applications for Windows, macOS, and Linux. However, because it is a desktop application, it must still be manually installed and updated by users and cannot be installed on operating systems such as iOS and Android.

Considering the actual usage scenario of BlueNavi (need to support as many platforms as possible to maximize compatibility with users' existing devices, but also to take into account the offline usage scenario and improve the user experience as much as possible, *etc.*), and the fact that the development of the Electron.js desktop application and the web application can basically share the same codebase, both the web and desktop versions of BlueNavi are developed.

### Angular Framework

As the saying goes, *there is no need to reinvent the wheel*; most of the time, people do not have to build their applications from scratch.

The Angular framework is a standardized set of concepts, practices, and criteria for dealing with a common type of problem that can be used as a reference to help developers approach and solve new problems of a similar nature [42]. It provides a standardized set of design patterns, project structures, and code styles that make code more readable and maintainable.

In frontend development, Angular, React, and Vue are the three most popular frameworks. While there is no *best* framework among them, Angular is a full-fledged Model-View-Controller (MVC) styled framework that provides clear guidance on how the application should be structured. It is maintained by Google, is open source, and supports bi-directional data flow while providing a real Document Object Model (DOM).

The Angular architecture includes models, components, templates, metadata, services, *etc.*, and their relationship is shown in Figure 4.2. Angular's comprehensive documentation provides an in-depth explanation of the framework's concepts and features, making it easy for developers to get started and leverage its capabilities.

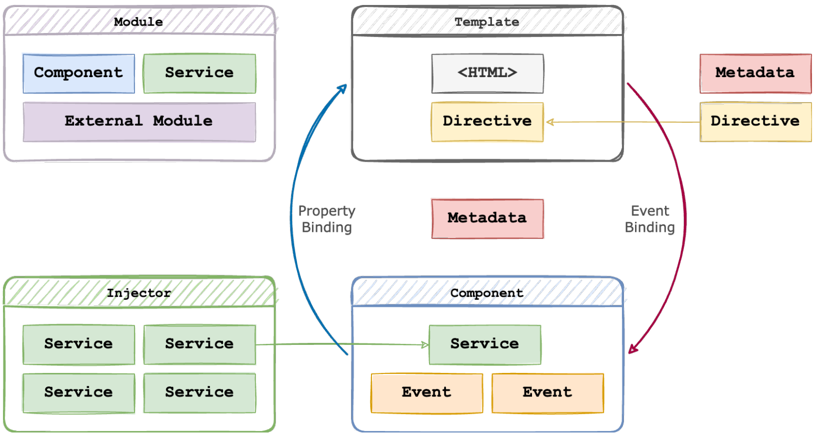


**Figure 4.2:** Angular framework [43].

### 4.1.3 Design and Implementation

The BlueNavi application has a user interface consisting of a menu bar and a map interface, as shown in Figure 4.3. The menu bar provides access to functions of the application, which are displayed in pop-up windows. The map interface is the main component for viewing and interacting with data.

The map interface includes layers for the map, chart, and fetched AIS data. The bottom layer consists of map data from third-party APIs. Users can select different types of maps, such as normal, topographic, or satellite, provided by various organizations. For example, users can overlay the visualized AIS data on Google Maps. Third-party map data are generally free to use, but access to them may require an internet connection.

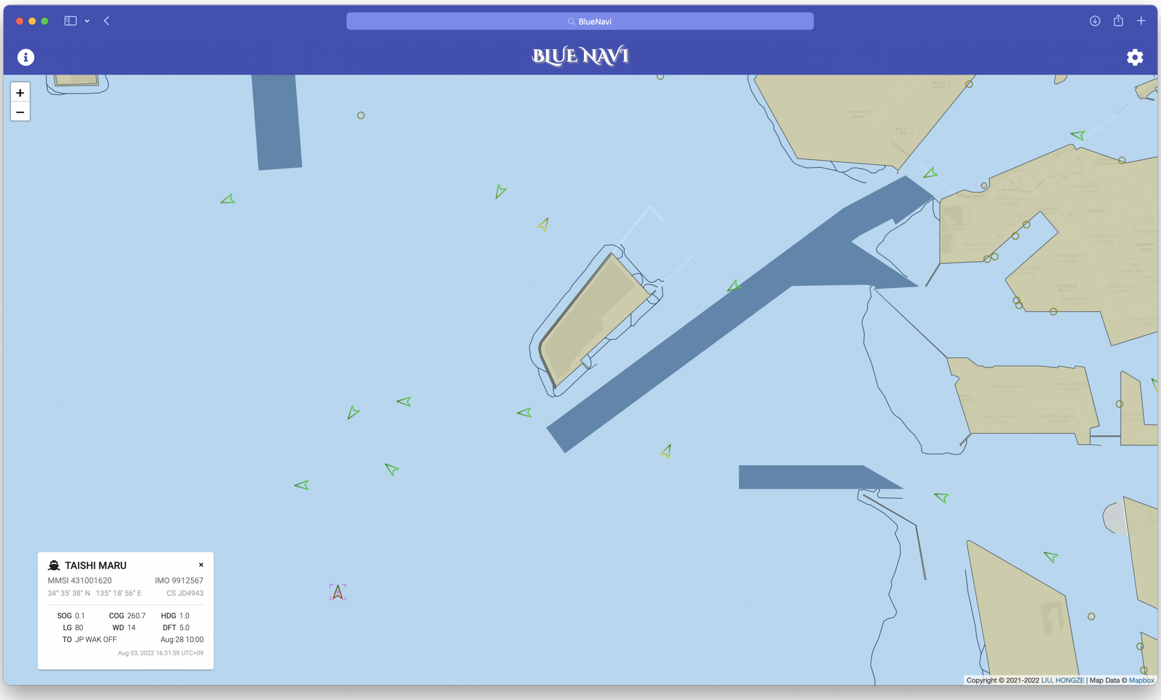
The BlueNavi application also supports the display of simple nautical charts in GeoJSON format if the user does not have an internet connection[3] or if the user is not satisfied with the map data provided by third-party organizations[4]. Figure 4.3 shows that the system displays the simple nautical chart data provided by the Japan Coast Guard (JCG). This chart data accurately shows navigational information, such as coastlines, breakwaters, and waterways, that are not available on land-oriented maps. The chart data is displayed on top of the map layer.

The third layer of the map interface displays the visualized AIS data, and the fourth layer is used to display other information. Each marker is bound to a click event. When the user clicks on a marker, additional information is displayed.

[3] While IP communication is commonly associated with the internet, it can also take place over a LAN.

[4] Because these map data are often not intended for nautical use.

As discussed in Section 4.1.2, web technologies are not limited to developing web applications but can also be used to create desktop applications. Figure 4.4 shows a desktop version of the BlueNavi application, which uses almost the same codebase as the web version.



TAISHI MARU ×
MMSI 431001620 IMO 9912567
34° 35' 38" N 135° 18' 56" E CS JD4943
SOG 0.1 COG 260.7 HDG 1.0
LG 80 WD 14 DFT 5.0
TO JP WAK OFF Aug 28 10:00
Aug 03, 2022 16:31:59 UTC+09

**Figure 4.3:** BlueNavi UI (web application).

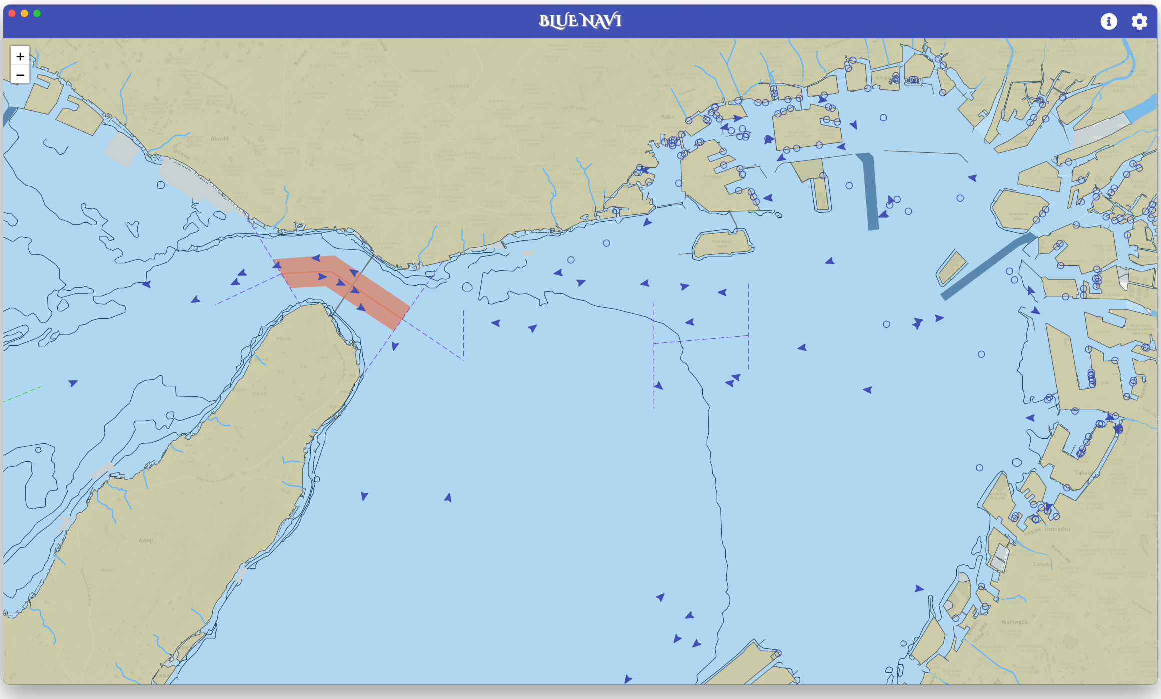


**Figure 4.4:** BlueNavi UI (desktop application).

## 4.2 RedNavi: Building 3D Scenes

In Section 4.1, an application was developed using web technologies to display One-Dimensional (1D) navigational information on a map, transforming it into more visually intuitive 2D information. This section goes one step further and explores other visualization methods for AIS information.

### 4.2.1 Background and Scenarios

In the BlueNavi system, navigational information is presented in a 2D format. However, novice mariners may find it challenging to correlate these 2D displays with the complex real-world 3D environment without extensive on-the-job experience [44]. For this reason, a 3D system can serve as a helpful bridge to expedite the development of these skills for less experienced mariners.

Compared to 2D graphics, 3D models offer several advantages: they are more visually efficient, easier to interpret, can provide more detailed information, and have the potential to increase users' awareness of their surroundings and understanding of the objects they represent [45]. Consequently, 3D visualization solutions have been extensively researched in different fields as diverse as the humanities and arts [46], public health [47], and MET [48–50]. Drawing inspiration from these studies and taking into account the unique features of AIS data, *RedNavi* has been developed as a web application that generates 3D scenes on the computer screen. These scenes are based on information about the own ship and AIS data from other ships, and provide a representation of both the environment and traffic.

Mariners rely on 2D AIS information during navigation, and correlating this information with real-world scenarios is a skill that typically takes time and experience. The RedNavi web application generates 3D representations of the environment and vessel traffic based on AIS data, allowing novice mariners and students to develop this skill during training. Moreover, computer-generated 3D scenes remain unaffected by adverse weather conditions or low visibility, providing valuable assistance to lookouts and enhancing situational awareness.

### 4.2.2 Tooling and Frameworks

RedNavi and BlueNavi share a common approach to data formatting and utilize similar web technologies. These similarities between the two applications enable RedNavi to inherit a number of features from BlueNavi. As discussed in Section 4.1.2, the use of a framework can significantly speed up the development process, which is why the same framework, Angular, is chosen for RedNavi as for BlueNavi.

The Three.js JavaScript library is utilized for rendering and presenting the 3D models. It is a well-established and widely used library for creating 3D graphics using web technologies. Three.js is built on top of the Web Graphics Library (WebGL) and provides a user-friendly set of APIs that abstract away the complexities of working with WebGL. With

Three.js, developers can easily create and display 3D graphics on web pages, enabling them to build rich and interactive experiences for their users.

### 4.2.3 Design and Implementation

The user interface of RedNavi is very similar to that of BlueNavi, featuring a menu bar, but with a 3D interface instead of a 2D map, as shown in Figure 4.5. The Three.js JavaScript library (*i.e.*, the WebGL engine) handles the rendering of the entire 3D interface.

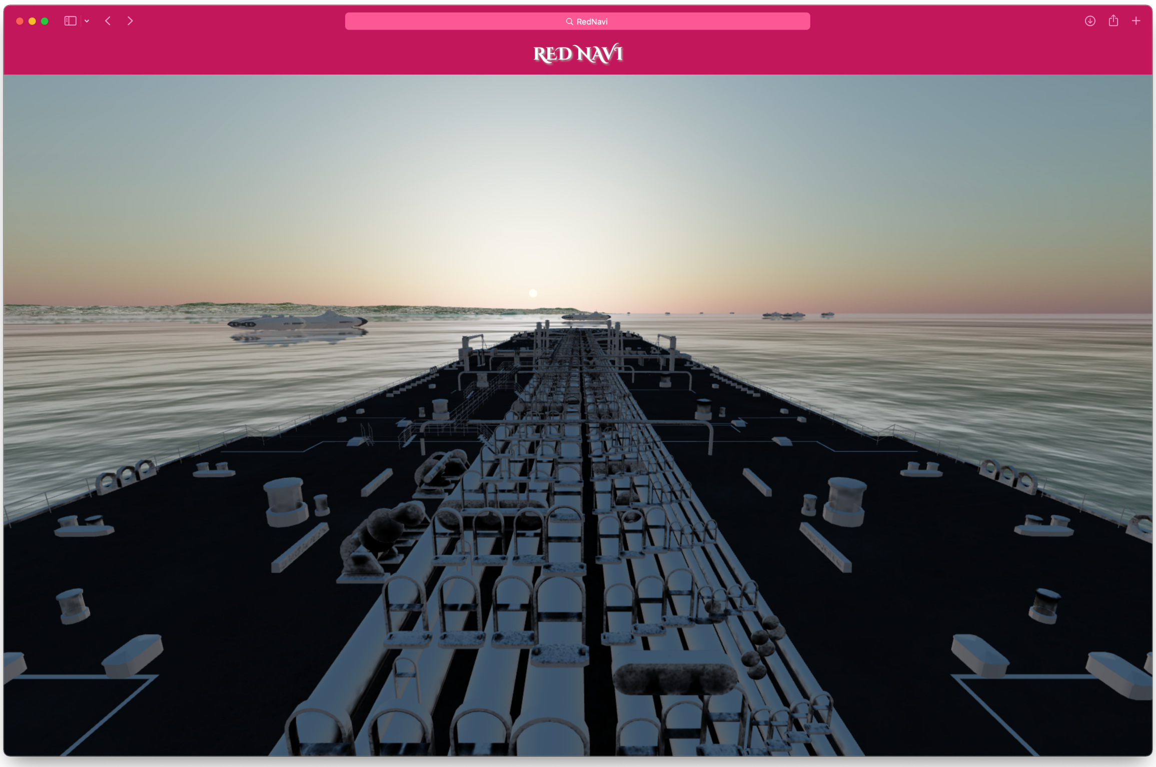


**Figure 4.5:** RedNavi UI: first-person view.

RedNavi's interaction with the backend follows the same logic as BlueNavi. Therefore, the development of RedNavi involved the reuse of BlueNavi's models and services[5]. However, the frontend of RedNavi has its unique processing flow once the application fetches data from the server. The system operation relationship diagram is depicted in Figure 4.6. The application distinguishes the own ship from other ships based on the Maritime Mobile Service Identity (MMSI) before rendering the terrain, updating the sun's azimuth and elevation, and other geographic location-dependent properties. The system then renders the ship at the origin, calculates the relative positions and headings of other ships based on the AIS information, and updates their headings accordingly.

After all the processing is complete, the user can interact with a 3D scene of the current sea surface. The scene view is not fixed, and the user can modify it using a mouse, keyboard, touch screens, *etc.* For example, it's possible to change the viewpoint of the scene, as shown in Figure 4.7.

[5] Here, the term *service* refers to the concept within the Angular framework, and not to a microservice in the backend.

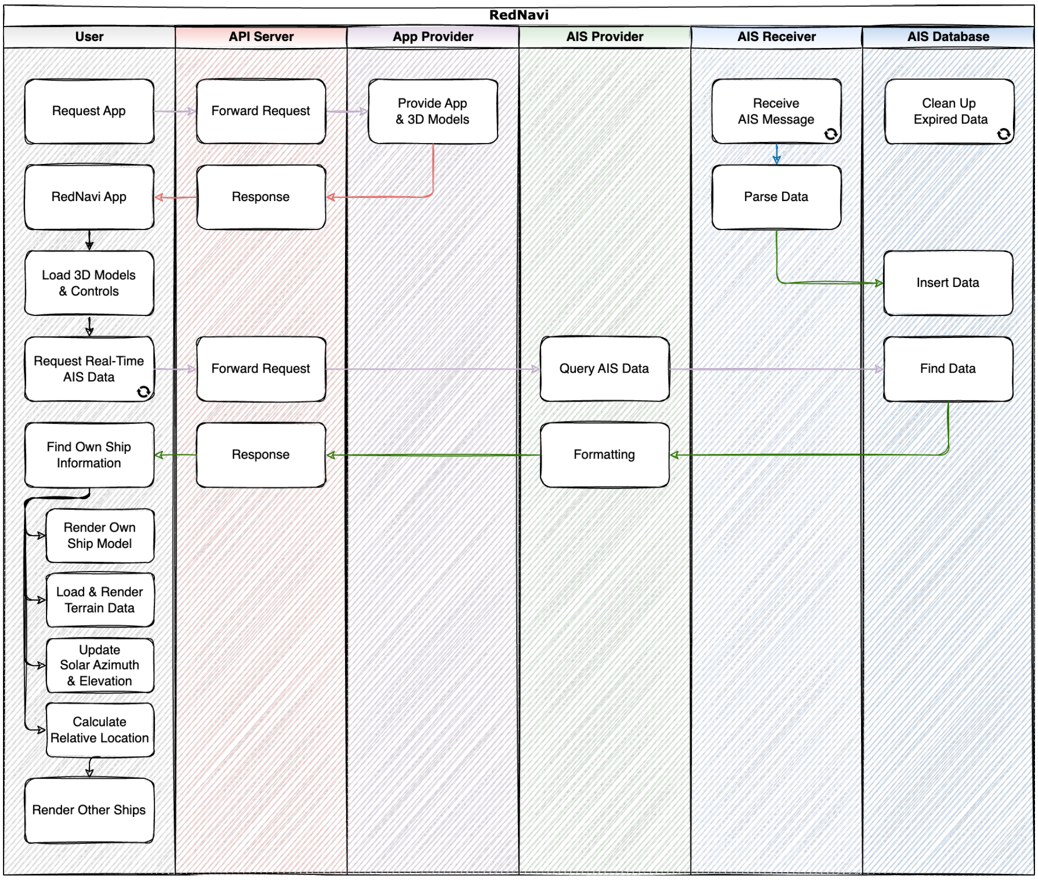


Figure 4.6: RedNavi workflow.

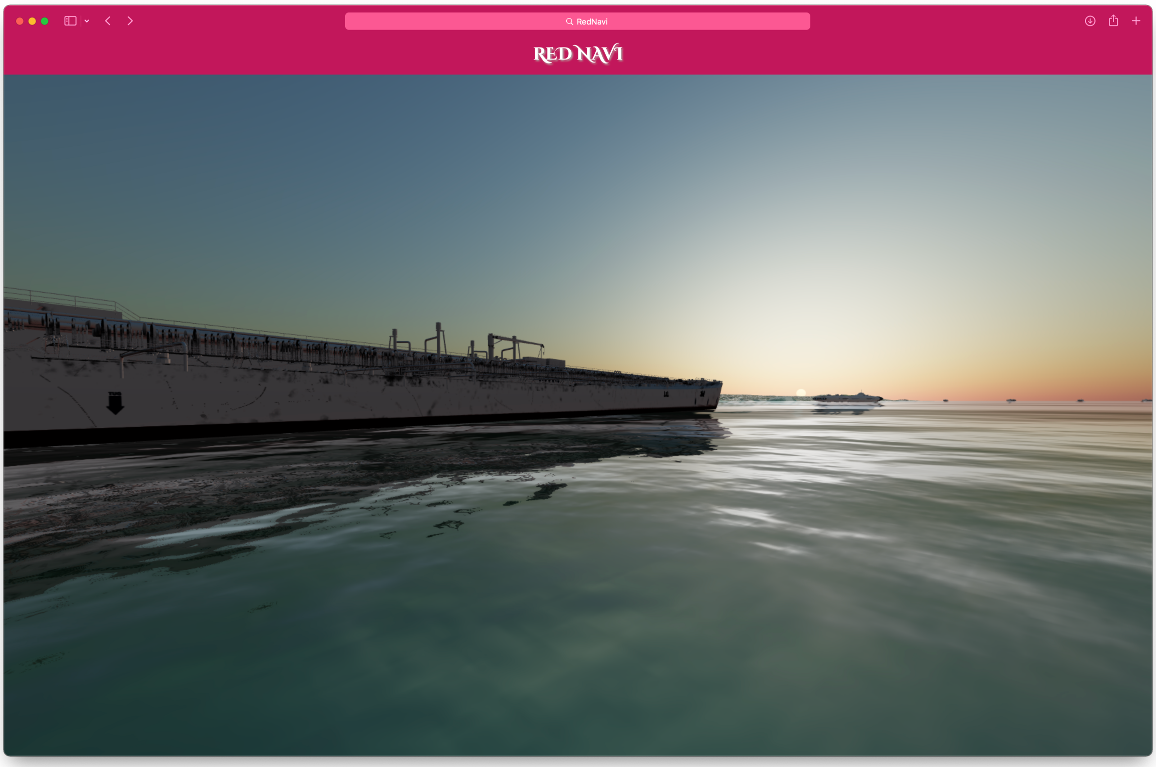


Figure 4.7: RedNavi UI: third-person view.

## 4.3 GreenNavi: Tracing Historical Data

Although AIS was originally designed to help ships avoid collisions at sea, the users of AIS are now far from being limited to mariners. In this section, another application is developed using web technologies to meet the needs of land users for AIS data.

### 4.3.1 Background and Scenarios

The IMO lists three main objectives of AIS: improving the preservation of the maritime environment, ensuring safety of life at sea, and enhancing navigation efficiency [18]. In addition to identifying ships and tracking targets, AIS supports SAR operations, reduces the need for verbal ship reporting, and enhances situational awareness. While these goals primarily benefit Officer of the Watch (OOW) onboard ships, the use of AIS data extends far beyond the maritime industry. Maritime pilots, shipping companies, port and traffic management departments, and others rely on AIS data for a variety of purposes, such as tracking ships, managing fleets, evaluating new projects, and performing statistical analysis of waters. Thus, AIS data has become an essential tool in all aspects of the maritime industry.

The various usage scenarios for AIS data, as described above, show that unlike ship collision avoidance, which requires real-time data with a high degree of timeliness, these scenarios typically rely on historical AIS data from the ship or water. Historical data is particularly useful for statistical analysis and other tasks.

With this in mind, the value of historical AIS data was thoroughly considered during the initial design of the entire system. A historical data database was designed to store obsolete data received, rather than discarding it directly. In addition, microservices for statistics were designed to analyze and process the historical data, and specific APIs were created to obtain these traces or statistics. To fully utilize the AIS historical data, the third web application, named *GreenNavi*, was designed and developed in this section.

### 4.3.2 Tooling and Framework

Although GreenNavi's frontend can use the Angular framework like BlueNavi and RedNavi, this dissertation opted to use the React library for frontend development in this section. React has gained widespread popularity and has become a go-to choice for web developers. Compared to Angular, React has some key differences and features that are worth noting.

**Library** In this chapter, the term *framework* is used to describe Angular, while *library* is used to describe React. This is because Angular comes with many functional libraries for different tasks, making it possible to do almost anything with Angular alone. React, on the other hand, focuses solely on building reusable UI components and requires other libraries, such as Redux for state management and React Router for route management, to build a full-featured application. This gives React more flexibility, but it can also make it more dependent on other libraries.

**One-Way Data Binding** React uses one-way data binding, meaning that changes to UI components do not affect changes to the state. In contrast, Angular uses two-way state binding, where the same data is updated across both the HTML element and the model variable, allowing changes to be made to the component's state as well as being displayed. Figure 4.8 illustrates the difference between one-way and two-way data binding.

**Virtual DOM** React uses a *virtual DOM* that generates a virtual representation of the entire user interface whenever changes are made. This virtual DOM is then compared to the previous virtual DOM to identify the differences, which are then rendered onto the real DOM, as shown in Figure 4.9. This approach significantly improves performance. In contrast, Angular uses *incremental DOM*, which converts each component into a set of instructions to build and modify the DOM tree, as shown in Figure 4.10. This reduces memory usage but is not as efficient as the virtual DOM.

Other differences between React and Angular include the languages used — Angular uses traditional HTML and JavaScript, while React uses JSX syntactic sugar[6]. However, these differences do not determine the *best* frontend solution, and React was chosen in this chapter to explore a different development approach.

[6] JSX is a syntax extension for JavaScript that allows HTML-like elements and components to be written in JavaScript code.

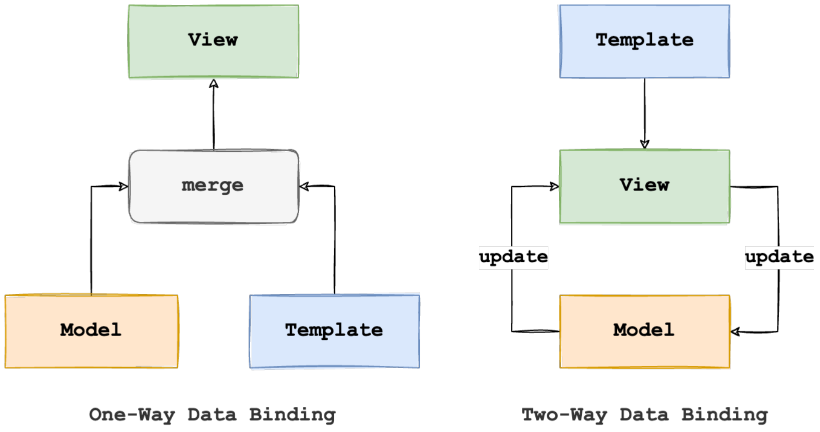


**Figure 4.8:** Data binding.

### 4.3.3 Design and Implementation

This subsection focuses on using React to develop the frontend of GreenNavi, which shares a similar layout with BlueNavi and RedNavi, with some UI design changes. As shown in Figure 4.11, the data displayed in GreenNavi is sourced from the backend API that retrieves historical data from the historical database. In addition to using the settings provided in the menu, GreenNavi also allows users to query historical data directly from the URL. For example, to view sea conditions at 20:00 on November 20, 2020, Japan Standard Time (JST), a user can either open the settings interface to set the target date and time or send a request as shown in Request 4.1 by typing the URL directly into the address bar of the browser.

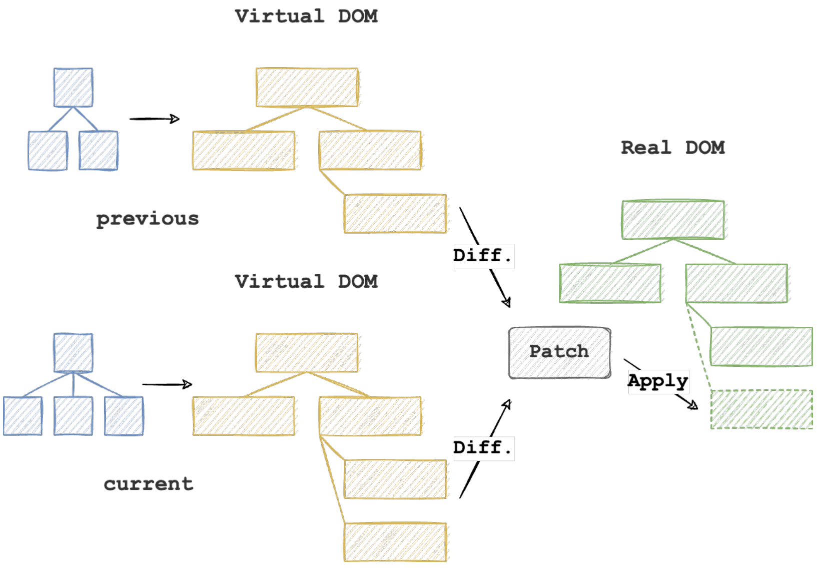


**Figure 4.9:** Virtual DOM.

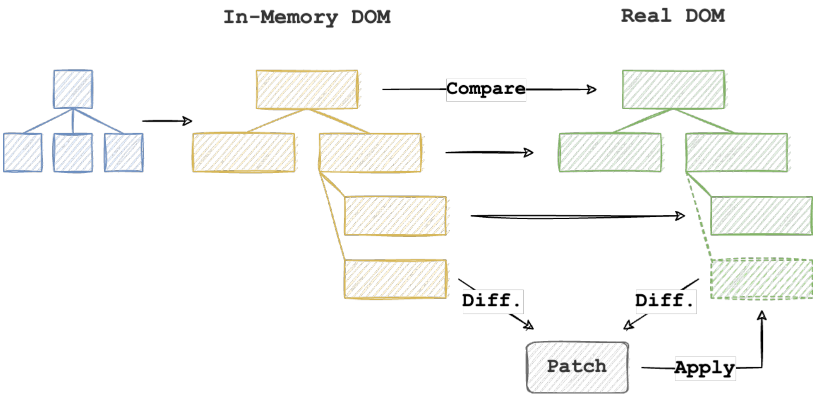


**Figure 4.10:** Incremental DOM.

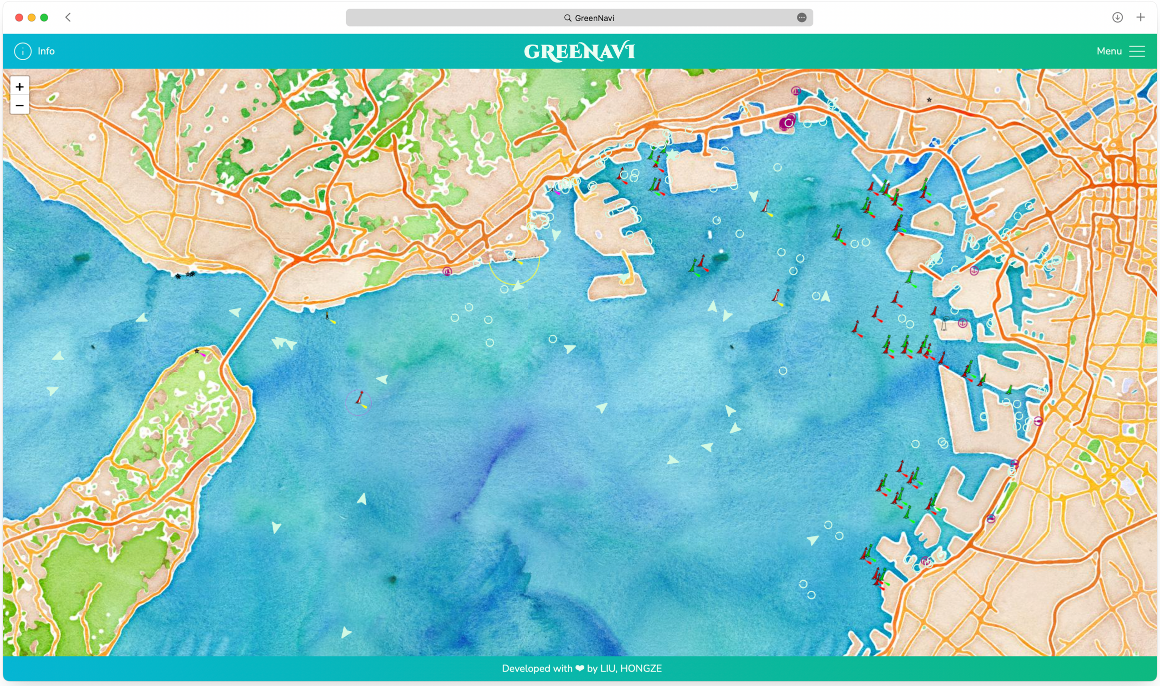


**Figure 4.11:** GreenNavi UI: historical scene.

**Request 4.1: Example request to the `/history` endpoint.**

```
GET https://{{baseUrl}}/history/at/2020/11/20/2000
Accept: */*
```

In this case, since the HTTP request sent by the user from the browser is aimed at getting the entire page of the display, not only the data themselves, the request here is different from the requests in Chapter 2. This feature is called routing and requires React to work with other libraries, *e.g.*, React Router. The response returned by the server is shown in Response 4.1.

**Response 4.1: Example response to the `/history` endpoint.**

```
HTTP/1.1 200 OK
Content-Type: text/html
<!DOCTYPE html>
<html>
    <head>
        <meta charSet="utf-8" />
        <meta name="viewport" content="width=device-width" />
        <title> GreenNavi </title>
        ...
    </head>
    <body>
        ...
    </body>
</html>
```

In addition to visualizing sea conditions, GreenNavi has the capability to display other types of historical or statistics data by utilizing designed APIs. For instance, Figure 4.12 shows the trajectories of the ship *YAHATA-MARU* throughout the year 2020. The use of historical navigation data can serve a critical function in ship management or port administration, as well as an aid to navigation. For example, in situations where two ships may cross paths, combining trajectory and statistical data with real-time AIS data can help the ship officer anticipate the movement of the other ship.

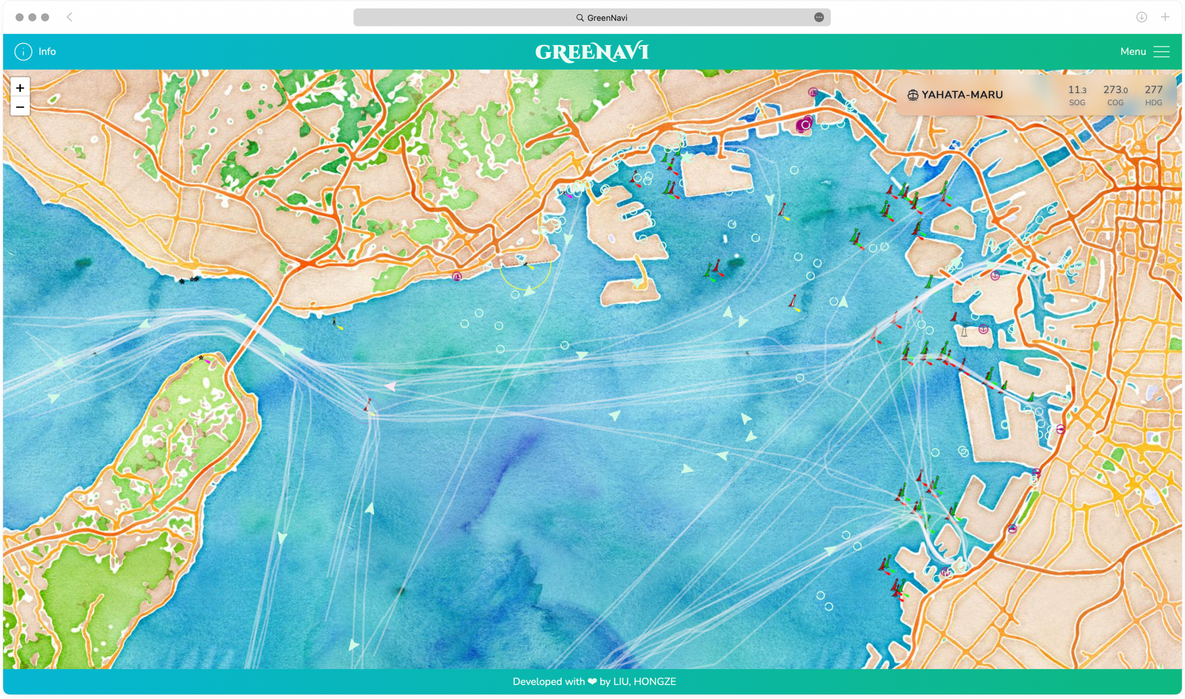


**Figure 4.12**: GreenNavi UI: trajectories.

## 4.4 Summary

In Chapter 4, the focus shifted to the implementation of the client side of the systems by designing and developing three different applications based on web technologies to cater to the varying data needs of the maritime industry. The first application, BlueNavi, was designed to provide real-time navigational information in a 2D form using AIS data. The second application, RedNavi, was designed to provide a visually intuitive 3D representation of navigational information. Finally, GreenNavi was developed as a demonstration of how web technologies can be used to satisfy the needs of users for historical data. Each of these applications was designed with a specific purpose in mind, but all three applications are the result of the integration of the system with microservices. The design and development of these applications demonstrate the potential for extending the functionality of modern maritime information support systems.

# Modern Communication Framework and Cross-Platform Applications

# Defining gRPC Services 5

In Section 2.3, inter-service communication was designed using the RESTful API based on the HTTP/1.x communication protocol, which is still the most widely used communication protocol. However, this protocol has some limitations that affect its performance. Therefore, starting from this chapter, gRPC will be explored as an alternative means of communication. gRPC is based on the next-generation HTTP protocol, HTTP/2, which offers several advantages over HTTP/1.x.

## 5.1 A Glimpse of gRPC

This section introduces gRPC and the HTTP/2 protocol on which it is based.

### 5.1.1 From RPC to gRPC

In Section 2.3, RESTful APIs were designed for inter-service communication. REST is an architectural style introduced by Roy Fielding in 2000 [51]. However, it was not the first inter-service communication method. This section examines the evolution of inter-service communication methods, starting with Remote Procedure Call (RPC). It also introduces gRPC, a contemporary inter-service communication technology that builds upon the RPC foundation and is based on the next-generation HTTP protocol, HTTP/2.

**RPC**

To understand the benefits of gRPC, it is important to first discuss RPC, the first inter-service communication method developed in the 1970s and 1980s. In non-distributed or monolithic systems, where all the code is deployed together, communication is typically achieved through function calls. The idea of RPC is to extend this concept to distributed or microservice architecture-styled systems, allowing remote services to be called as if they were local procedures, without worrying about the details of remote communication implementation. This model works by sending a request from the calling system (client) to the called system (server), which executes the request and returns the response, making it possible to create distributed systems where different components can reside on different machines while functioning as if they were on the same machine. While early implementations of RPC, such as the Common Object Request Broker Architecture (CORBA) and Java Remote Method Invocation (RMI), were complex and bloated in specification, they paved the way for the evolution of inter-service communication methods [52].

### REST

REST, which stands for Representational State Transfer, was first proposed by Roy T. Fielding in his doctoral dissertation in 2000 [51]. REST is based on the first generation of the HTTP protocol (HTTP/1.x) and is a popular architectural style for building web services. Unlike RPC, which uses verbs to describe endpoints (*e.g.*, `/getTrajectories`), REST uses several HTTP methods such as `GET` (for requesting), `POST` (for creating), `PUT` (for modifying), `DELETE` (for deleting), `PATCH` (for partial updating), *etc.* These methods are also known as HTTP verbs and are combined with endpoints to form a request (*e.g.*, `GET /trajectories`).

REST messages support various formats, including JavaScript Object Notation (JSON), Extensible Markup Language (XML), HTML, and YAML Ain't Markup Language (YAML). Among them, JSON has become the de facto format for building microservices due to its widespread support and ease of use. However, despite being a human-readable plain text format, JSON also have performance implications.

In addition to the challenges associated with the HTTP/1.x protocol, which will be discussed in detail in Section 5.1.2, the REST architectural style itself can be problematic due to its high degree of flexibility. While flexibility can be considered a strength that gives designers and developers with the freedom to create and develop APIs, excessive flexibility can lead to a number of issues. For example, using `DELETE /trajectories` to insert trajectory data and `GET /dynamics` to delete navigational data is possible with the REST architecture, despite being extremely counterintuitive. In other words, the REST architectural style is not rigid and is challenging to enforce.

Moreover, the flexibility offered by REST can lead to weak interface binding between services. Unlike other communication styles, RESTful interfaces do not require up-front definition, and their implementation is not strictly standardized. For instance, when using the JSON format for data transfer, a vessel's unique identifier, the MMSI, can be represented in a variety of ways, such as `{ MMSI: 123456789 }`, `{ mmsi: "123456789" }`, `{ UserID: 123456789 }`, `{ user_id: "123456789" }`, *etc.* While all of these representations are valid, the service consumer has no way of inferring the response definition unless it is agreed upon during the service's design phase of the service. Unfortunately, this definition is optional and flexible, often leading to compatibility issues and bugs if not carefully managed. Although some third-party tools have emerged for defining RESTful APIs have emerged, such as the Open API used in our systems, they are still considered optional, not mandatory.

### gRPC

gRPC is an open source RPC framework released by Google[1]. It offers a combination of the benefits of RPC and REST, and has quickly gained popularity. gRPC offers several features and benefits, including:

**High Performance** gRPC uses a binary protocol called *protocol buffers* to transfer data, which is more efficient than text-based protocols. gRPC's protocol is based on HTTP/2, which further improves its performance.

[1] Despite being developed by Google, the letter *g* in gRPC does not stand for Google, but rather has different meanings in different releases. For example, in the first version, 1.0 of gRPC, the *g* stands for *gRPC* itself, and in subsequent versions such as 1.1, 1.2, and 1.3, the *g* stands for *good*, *green*, and *gentle*, respectively. In the latest version, 1.51 (as of February 2023), the *g* stands for *galaxy*. To learn more about the meaning of *g* in each version of gRPC, see Reference [53].

**Emphasis on Definition** gRPC requires the developer to first define the service interface and then implement it in code. All microservices, including client and server implementations, must strictly adhere to this service definition to avoid compatibility issues.

**Strongly Typed Messages** gRPC supports messages with strongly typed data, where the data type is included in the message definition. This makes communication more predictable.

**Cross-Language, Cross-Platform Support** gRPC is not tied to a specific programming language. During the design phase, only the message and service interfaces need to be defined, and the gRPC code generator can generate the necessary implementation code for different languages and platforms. Currently, gRPC supports a wide range of programming languages and platforms, including C#/.NET, C++, Dart, Go, Java, Kotlin, Node.js (JavaScript), Objective-C, PHP, Python, Ruby, *etc.*

**Streaming Support** gRPC natively supports streams, which means that it can send and receive multiple messages in a single RPC call. This is one of the main advantages of gRPC.

gRPC is now used by many well-known companies and organizations, due to its many benefits and features.

### 5.1.2 gRPC over HTTP/2

While RESTful APIs are typically built on top of the widely adopted HTTP/1.x protocol, this protocol still suffers from several drawbacks, including [54]:

**Network Latency** With HTTP/1.x, multiple open TCP connections may cause network latency.

**Head-of-Line (HOL) Blocking** HTTP/1.x can only handle one request per connection at a time. If a request gets blocked (*e.g.*, due to output congestion), subsequent requests have to wait.

**Redundancy of Headers** In HTTP/1.x, headers such as User-Agent are sent repeatedly over multiple requests, resulting in redundant data being sent over the wire and potentially wasting bandwidth.

**Text-Based Protocol** HTTP/1.x is a text-based protocol that uses a human-readable format (*e.g.*, JSON or XML) to transmit requests and data. While this makes it convenient for developers, it also consumes transmission bandwidth, degrades performance, and introduces potential security issues.

To address the issues with HTTP/1.x, HTTP/2 was proposed as the second major version of the HTTP protocol to provide a more efficient and secure protocol for web communications. HTTP/2 supports all of the core features of HTTP/1.x, but is more efficient, simpler, and more robust [55]. It has several key features and benefits:

**Binary Protocol** Using protocol buffers as the default payload format in gRPC means that less data needs to be transmitted over the network, reducing bandwidth usage and improving overall system performance. In a modern maritime information support system, this can be particularly useful when transmitting large amounts of real-time data, from navigational information such as ship locations and meteorological information such as weather charts, to media such as audio and video.

**Bidirectional Data Flow** HTTP/2 introduces the concept of streams, allowing communication to occur in both directions in *frames*, rather than in a one-to-one request-response mode, once a TCP connection is established between the client and server. gRPC supports bidirectional streaming, allowing both clients and servers to send and receive multiple messages in real-time. This can be useful in a modern maritime information support system where there is a need for real-time communication between different vessels or with shore-based stations.

**Multiplexing** HTTP/2 divides messages into separate frames, which is the smallest unit of data transmittion. HTTP/2 achieves multiplexing by sending data in these frames instead of the entire message. Multiplexing allows server-client communication over a single TCP connection without the HOL blocking as of HTTP/1.x by eliminating the need for additional connections. This can be especially useful in a modern maritime information support system with multiple devices and systems that need to communicate with each other.

**Header Compression** HTTP/2 uses HPACK for header compression, which is secure and efficient. HPACK uses static Huffman coding to compress headers, which reduces the size of the header data being transmitted, further reducing bandwidth usage and improving performance [56]. In a real-world application scenario of a modern maritime information support system, where bandwidth may be limited, this can be a critical feature.

**Stream Prioritization** HTTP/2 allows developers to optimize the order of data transfer by constructing a priority tree, enabling critical data to be transmitted first and less critical data to be transmitted later, as opposed to the sequential transfer of HTTP/1.x. Data with higher priority can be transferred first. This can be useful in a modern maritime information support system where some data, such as distress signals, must be transmitted with the highest priority.

**Server Push** With HTTP/2, the server can intuitively push data, skipping the roundtrip instead of waiting for the client to request. The client has the option to accept or reject the server push. In a modern maritime information system where time is critical, this can be a valuable feature.

### 5.1.3 gRPC Communication Patterns

One of the key advantages of gRPC is its support for a variety of communication patterns beyond simple request-response. With gRPC, both the client and server can use streaming semantics to send a stream of messages in a single RPC call, rather than one message at a time. gRPC supports four types of communication patterns: unary (simple) RPC, server-streaming RPC, client-streaming RPC, and bidirectional streaming RPC [52]. This flexibility enables gRPC to handle a wide range of use cases, from simple point-to-point communication to complex, real-time applications.

### Unary RPC

Unary RPC is a straightforward communication pattern of gRPC, similar to REST. The gRPC stub sends a request to the gRPC server, and the server returns a response. Figure 5.1 shows a diagram of the unary RPC.

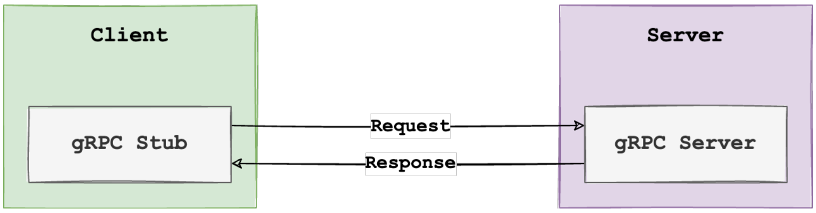


**Figure 5.1:** Unary RPC.

### Server-Streaming RPC

Unlike unary RPC, server-streaming RPC returns a sequence of responses after the client sends a request to the server. For example, the server can send all corresponding real-time data about the current sea state to the client as a stream. When there is no more data, the gRPC server sends a trailer to the gRPC stub to close the connection. Figure 5.2 shows a diagram of the server-streaming RPC.

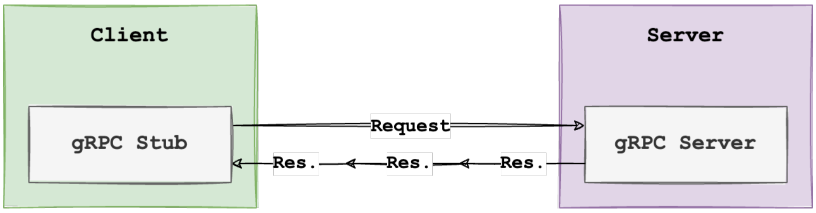


**Figure 5.2:** Server-streaming RPC.

### Client-Streaming RPC

Client-streaming RPC, on the other hand, is the opposite of server-streaming RPC. The gRPC stub sends a sequence of requests to the gRPC server, and the server can return a response to the client at any time, not just after receiving all the messages. After sending all messages, the client sends an End of Stream (EOS) flag to the server to close the connection. Figure 5.3 shows a diagram of client-streaming RPC.

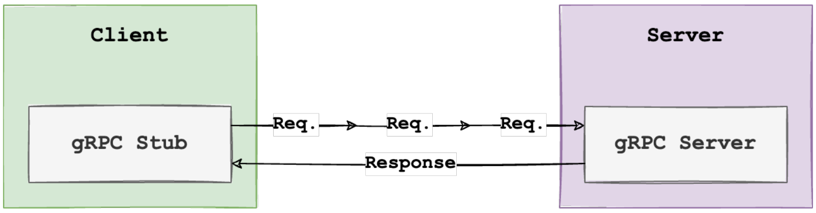


**Figure 5.3:** Client-streaming RPC.

### Bidirectional streaming RPC

In bidirectional streaming RPC communication, the gRPC stub sends a message stream to the gRPC server, and the server returns a message stream back to the stub. Bidirectional streaming communication must first be initiated by the client, and the subsequent communication depends on the business logic design of the system. Figure 5.4 shows a diagram of bidirectional streaming RPC.

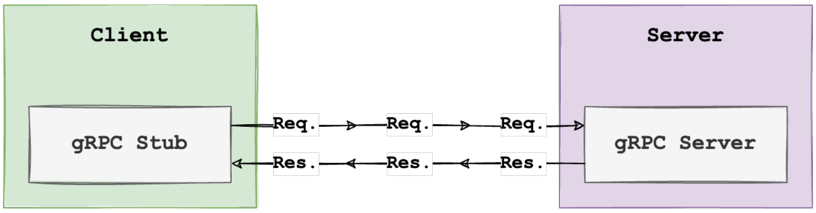


**Figure 5.4:** Bidirectional streaming RPC.

## 5.2 gRPC Services Design

To effectively design a modern maritime information support system using gRPC, it is essential to prioritize interface design. This section will define the service interfaces and messages from the server-side perspective, with the primary hypothetical user being the shipboard user. Three scenarios for the system will be considered: providing data to the client, collecting data from the client, and exchanging data with the client. The implementation of these service interfaces and messages will be described in detail in the next chapter.

### 5.2.1 Providing Maritime Information

Providing information from the server to the client is a primary function of a modern maritime information support system. In the case of the new system, users can query or subscribe to receive updates on various information.

In Section 4.1, BlueNavi was developed to cater to the needs of small vessels without onboard AIS equipment by providing AIS data through a backend server. The frontend application then uses the data to plot it on the map. The server uses RESTful APIs to provide the data. However, this communication mechanism has certain limitations. The client has to send requests to the server at regular intervals, which can result in delays in receiving the latest data. On the other hand, sending data at too short intervals can lead to redundant data transfer, which results in wastage of resources.

Using a stream-supported gRPC communication solves these problems. With gRPC, the client sends a request to the server, which returns all current data in the real-time database as a stream. Any changes (events) that occur in the database are then automatically sent back to the client via the same connection, provided that the request conditions are met. This approach ensures that the client always receives the latest real-time data and minimizes redundant data transmission, making the system more efficient.

### Providing a Single Point of Query

A gRPC service is essentially a group of remotely invokable methods. Defining a service using protocol buffers involves two key steps: defining the service methods and defining the messages used within the service.

If the client wants to make simple requests, unary RPC is the ideal choice. For example, if a client wants to retrieve static information about a particular ship based on its MMSI just once, the service can be defined as shown in Listing 5.1.

```
service Static {
  rpc getStatic(StaticDataRequest) returns (StaticData) {};
  ...
}
```

**Listing 5.1**: Definition of gRPC service `Static`.

In Listing 5.1, the `service` keyword is used to specify a named gRPC service `Static` that contains an RPC method `getStatic` specified by the `rpc` keyword. With the `getStatic` method, the client can send a request message `StaticDataRequest` using the stub and receive a response message `StaticData` from the server. The `StaticDataRequest` and `StaticData` messages are defined as in Listings 5.2 and 5.3, respectively.

```
message StaticDataRequest {
  string cID = 1;
  int32 uID = 2;
}
```

**Listing 5.2**: Definition of message `StaticDataRequest`.

```
message StaticData {
  optional string _id = 1;
  int32 mID = 2;
  int32 uID = 3;
  int64 tSt = 4;
  optional int32 imo = 5;
  string vNm = 6;
  string cSg = 7;
  int32 typ = 8;
  repeated int32 dim = 9;
  optional string dst = 10;
  optional double dft = 11;
  repeated int32 ETA = 12;
}
```

**Listing 5.3**: Definition of message `StaticData`.

Listings 5.2 and 5.3 define the named messages `StaticDataRequest` and `StaticData` using the keyword `message`. The `StaticDataRequest` message consists of two fields:

**`cID`** Client ID, of type `string`;
**`uID`** User ID (MMSI), of type `int32` (32-bit integer).

The `StaticData` message consists of twelve fields, each with a unique identification number used to identify the fields when the data is transmitted in binary format[2]. The keyword `optional` in protocol buffers indicates that a field is not required and may be omitted from the message, and `repeated` indicates that a field may occur zero or more times in a message and is typically used to transfer arrays or lists of values.

**`_id`** ID of the document in the database, of type `string`;

[2] In gRPC, the field identification number is actually more significant than the field name in binary format messages. Consequently, if a field is required to be deprecated due to changes in requirements, its identification number must typically be reserved using the `reserved` keyword and cannot be reassigned to other fields.

**mID** Message ID, of type `int32`;
**uID** User ID (MMSI), of type `int32`;
**tSt** Timestamp of the time the message was received, in milliseconds, of data type `int64` (64-bit integer);
**imo** IMO number, of type `int32`;
**cSg** Vessel call sign, of type `string`;
**vNm** Vessel name, of type `string`;
**typ** Vessel type, of type `int32`;
**dim** Vessel's dimensions, of type `int32`;
**dst** Voyage destination, of type `string`;
**dft** Maximum draft, of type `double` (double-precision 64-bit IEEE 754 floating point);
**ETA** Formatted estimated time of arrival, of type `int32`.

### Continuously Providing Data to the Client

When dealing with dynamic information, unary RPC may not be the best approach as data changes frequently and transmission redundancy should be avoided to ensure efficient data transfer. Listings 5.4 through 5.6 define the gRPC service `Dynamic`, the message `DynamicDataRequest`, and the message `DynamicData`, respectively:

**Listing 5.4**: Definition of gRPC service `Dynamic`.

```
service Dynamic {
  rpc getDynamics(DynamicDataRequest)
      returns (stream DynamicData) {};
  ...
}
```

This service includes the method `getDynamics`, which returns a stream of response messages `DynamicData` in response to a `DynamicDataRequest` from the client. The keyword `stream` is used to indicate that the response consists of multiple messages.

**Listing 5.5**: Definition of message `DynamicDataRequest`.

```
message DynamicDataRequest {
  string cID = 1;
  optional double lat = 2;
  optional double lon = 3;
  optional int32 RNG = 4;
}
```

`DynamicDataRequest`, as defined in Listing 5.5, has four fields:

**cID** Client ID, of type `string`;
**lat** Latitude of the request location, of type `double`;
**lon** Longitude of the request location, of type `double`;
**RNG** Range or radius of the area in which the user wants to request data, centered on the `[lat, lon]` coordinates, of type `int32`.

**Listing 5.6**: Definition of message `DynamicData`.

```
message DynamicData {
  optional string _id = 1;
  optional int32 mID = 2;
  int32 uID = 3;
  int64 tSt = 4;
  double lat = 5;
  double lon = 6;
  optional int32 nSt = 7;
  optional float SOG = 8;
  optional float COG = 9;
  optional int32 HDG = 10;
  optional int32 ROT = 11;
}
```

The response message `DynamicData` returned by the server includes the following eleven fields:

**`_id`** ID of the document in the database of type `string`;
**`mID`** Message ID, of type `int32`;
**`uID`** User ID (MMSI), of type `int32`;
**`tSt`** Timestamp of the time the message was received, in milliseconds, of data type `int64`;
**`lat`** Latitude, of type `double`;
**`lon`** Longitude, of type `double`;
**`nSt`** Navigation status, of type `int32`;
**`SOG`** Speed Over Ground, of type `float` (single-precision 32-bit IEEE 754 floating point);
**`COG`** Course Over Ground, of type `float`;
**`HDG`** True heading, of type `int32`;
**`ROT`** Rate of Turn (ROT), of type `int32`.

### Pushing Notifications to the Client

The server-streaming RPC enables the server to send information such as notifications, messages, and warnings to the client. The notification-pushing feature is defined using the same communication pattern as described above, so only a brief description is provided here.

Listing 5.7 gives the definition of the gRPC service `Notification`.

```
service Notification {
  rpc subscribeNotification(Subscription)
      returns (stream Notification) {};
  ...
}
```

**Listing 5.7**: Definition of gRPC service `Notification`.

To use this service, the client establishes a connection by invoking the `subscribeNotification` RPC method and sending the requested data in the `Subscription` message during initialization or as specified by the user. The connection is maintained by the client, and the server can push data to the client through the `Notification` message whenever necessary.

## 5.2.2 Collecting Maritime Information

Aside from transmitting data to the client, the server may also need to gather data from the client, with various types of data being possible. For instance, the ship reporting system commonly used in navigation practice requires eligible ships to report to the appropriate authority in a timely fashion, which includes detailed reporting requirements for entering and leaving ports at sea or on inland rivers in some countries or areas. Currently, the ship reporting system is moving towards digitalization and informatization, allowing reports to be completed online. In this section, RPC services will be designed to facilitate the reception and collection of data from the client.

### Receiving Static Reports

Referring to systems such as ship reporting as static reporting, since they only need to be reported once over a long period of time, unary RPC communication can be used without the need to maintain a constant connection to the client.

Part of the definition of the gRPC service `Report` is given in Listing 5.8.

**Listing 5.8**: Definition of gRPC service `Report` (static report).

```
service Report {
  rpc reportFromReportintLine(StaticReport)
      returns (StaticReportResponse) {};
  rpc reportFromReportingPoint(StaticReport)
      returns (StaticReportResponse) {};
  rpc reportFromAnchorage(StaticReport)
      returns (StaticReportResponse) {};
  ...
}
```

### Receiving Dynamic Reports

Dynamic data reports report data to the server at shorter intervals and may contain information such as geographic location, heading, speed, and other custom information from various sensors. While AIS can be used to transmit dynamic navigational information, there are reasons why it needs to be reported to the server via gRPC, which will be discussed in Chapter 6. The service interface for reporting such data is defined in Listing 5.9.

**Listing 5.9**: Definition of gRPC service `Report` (dynamic report).

```
service Report {
  ...
  rpc reportDynamic(stream DynamicReport)
      returns (DynamicReportResponse) {};
  ...
}
```

In contrast to static reports, the client sends a stream of `DynamicReport` messages to the server, as defined in Listing 5.10, using the client-streaming RPC.

**Listing 5.10**: Definition of message `DynamicReport`.

```
message DynamicReport {
  string cID = 1;
  int64 tSt = 2;
  double lat = 3;
  double lon = 4;
  float SOG = 5;
  int32 HDG = 6;
  optional int32 alt = 7;
  optional int32 hAcc = 8;
  optional int32 sAcc = 9;
}
```

Several non-optional fields in Listing 5.10 have the same meaning and data type as those in the previous AIS dynamic data (message `DynamicData` in Listing 5.6), and the remaining fields (`alt`, `hAcc`, and `sAcc`) are described as follows. A more detailed explanation of the dynamic report and its application will be given in Chapter 6.

**`alt`** Altitude, of data type `int32`;
**`hAcc`** Horizontal accuracy, of type `int32`;
**`sAcc`** Speed accuracy, of type `int32`.

### 5.2.3 Exchanging Maritime Information

Suppose that the user only needs information about a specific set of ships at sea, for example, all ships within a certain range or vicinity (*e.g.*, 6 nautical miles) from their current location. Moreover, the user's location is constantly changing as the ship navigates the waters. In this case, the client may need to include its location information (along with other query conditions) in each request message. The server then filters and queries the data according to these conditions and returns the results to the client. For such a scenario, bidirectional streaming RPC is a better communication method to use.

The RPC method `exchangeDynamics`, defined in Listing 5.11, is an example of the dynamic data-exchange RPC method that uses bidirectional streaming. This method involves the client continuously sending `DynamicReports`, containing its location information, to the request stream. The server then captures the necessary report information, includes its corresponding response in messages `DynamicData`, defined in Listing 5.6, and sends them to the response stream.

```
service Dynamic {
  ...
  rpc exchangeDynamics(stream DynamicReport)
      returns (stream DynamicData) {};
  ...
}
```

**Listing 5.11**: Definition of gRPC service `Dynamic` (exchange dynamics).

## 5.3 Summary

Chapter 5 focused on the use of gRPC as a means of communication for the modern maritime information support system. The chapter begins with an introduction to gRPC and the underlying HTTP/2 protocol. This is followed by the definition of the gRPC services for the system. The chapter outlined the three main scenarios for the gRPC services: providing data to the client, collecting data from the client, and exchanging data with the client. The implementation of these services will be described in the following chapter. The use of gRPC emphasizes the importance of interface design, making it a valuable addition to the system's inter-service communication.

# Implementing gRPC Services 6

Chapter 5 provided the service interface and messages for gRPC. As discussed in Section 5.1.1, traditional RPC can be cumbersome due to the need for manual specification and implementation. One of the major advantages of gRPC is that it supports multiple languages and platforms through code generation. After defining services and corresponding messages, gRPC generates code for the desired language or platform. This simplifies remote communication between microservices or clients and servers, as developers can call remote services in the same way they call local functions.

This chapter focuses on selecting appropriate tools and frameworks from both server-side and client-side perspectives and utilizing them to implement the services.

## 6.1 Implementing gRPC Server

Chapter 3 introduced several microservices based on the REST architecture, focusing mainly on functional implementation. In contrast, this section aims to improve performance and efficiency by implementing critical microservices using gRPC as the communication framework and a new language base. By leveraging gRPC's code generation capabilities, the system's microservices will have better multi-language and multi-platform support, leading to a more scalable and efficient microservice architecture.

### 6.1.1 Tooling and Framework

When it comes to backend development, there is no one *best* programming language. In this dessertaion, Node.js was chosen for its simplicity and versatility. As a unified platform for both frontend and backend development, Node.js offers many benefits, including ease of collaboration between frontend and backend teams due to its use of a shared language. Meanwhile, JavaScript is a dominant language in frontend development, making it a natural choice for many web applications. By using Node.js for backend development, developers can leverage the power of JavaScript and take advantage of a unified development ecosystem.

However, as a system scales, it is important to reconsider the choice of implementation tools, as Node.js presents some challenges:

**Dynamic Typing** Unlike some other strongly typed languages, JavaScript is dynamically typed. While dynamic typing provides flexibility, it also increases the likelihood of bugs, especially as a project grows.

**Asynchronous Programming** JavaScript uses an asynchronous programming model, which can provide benefits in terms of scalability. However, it also introduces issues such as the well-known *callback* *hell* problem. The complex asynchronous model can make large projects difficult to maintain.

**Excessive Dependencies and Instability** Node.js boasts a robust ecosystem and rich third-party libraries, but projects often rely on many third-party libraries with complex dependencies. This can lead to compatibility issues and additional maintenance costs as developers update versions, change, or deprecate APIs.

Given these challenges, a more suitable backend development language is desired. The Go programming language has garnered attention as a potential solution.

Go is a relatively new programming language developed by Google in 2007 and first released in 2009. It was designed to avoid the problems of previous programming languages while incorporating several advantages [57]. Here are some of Go's main strengths:

**Simplicity** Go's simplicity comes from its neat and clean syntax, which is a departure from the simplicity of Node.js. For example, the Go language has only 25 keywords, which increases the clarity and readability of the code. This simplicity makes Go code easier to develop and maintain.

**Performance** As a compiled language, Go has a performance advantage over many other high-level languages because it can be compiled directly into machine language. In contrast, JavaScript is an interpreted language that must be interpreted by the browser at runtime. However, Go also has the features of a high-level language, such as garbage collection.

**Concurrency** Go was designed from the ground up to support concurrent computing and processing. *Goroutines* enable functions and methods to execute in parallel, but with an appropriate communication mechanism, sharing memory and avoiding competition. In contrast, Node.js supports concurrency through its event callback mechanism, but it is still essentially a single-threaded tool where only one function or method can occupy computing resources at a time. Consequently, its support for concurrency is inferior to that of Go.

Go is an open-source, statically-typed, multi-purpose programming language that is gaining popularity due to its simplicity, performance, and concurrency support. Its cross-platform capability also makes it an attractive choice for developing applications that can run on different operating systems.

In this dissertation, since the backend services of a modern maritime information support system may require high performance and concurrency to handle the large volume of data parsing, computing, and processing required while serving multiple users simultaneously, Go was selected as the implementation language for some specific microservices to take advantage of its concurrency support and efficient memory management.

### 6.1.2 Implementation

As discussed earlier, gRPC is a cross-platform framework that supports multiple languages, including Go, and provides code-generation capabilities. Once Go is selected as the language for implementing gRPC-related microservices and the gRPC service definition is complete, the code required to implement the service interface can be easily generated using simple commands.

This subsection uses the `Dynamic` service defined in Chapter 5 as an example. The `protoc` command for generating the code is shown in Listing 6.1, which generates `interfaces`, `structs`, and other code components in Go based on the associated `.proto` file for the `Dynamic` service definition.

```
protoc --proto_path=proto --go_out=./pbs/dynamic \
--go-grpc_out=./pbs/dynamic protos/dynamic/v1.dynamic.proto
```

**Listing 6.1**: Command for generating code in Go.

Running the protoc command for the `Dynamic` service will generate two files: `v1.dynamic.ais.pb.go` for message-related code and `v1.dynamic.ais_-grpc.pb.go` for gRPC communication-related code. Listings 6.2 and 6.3provide examples of the code generated in these two files.

```
// Code generated by protoc-gen-go. DO NOT EDIT.
// versions:
//    protoc-gen-go v1.28.1
//    protoc        v3.21.5
// source: ais/v1.dynamic.ais.proto

package ais

import ...

const (...)

type DynamicDataRequest struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    CID  string   `protobuf:"bytes,1,opt,name=cID,proto3,
          oneof" json:"cID,omitempty"`
    Lat  *float64 `protobuf:"fixed64,2,opt,name=lat,proto3,
          oneof" json:"lat,omitempty"`
    Lon  *float64 `protobuf:"fixed64,3,opt,name=lon,proto3,
          oneof" json:"lon,omitempty"`
    RNG  *float64 `protobuf:"fixed64,4,opt,name=RNG,proto3,
          oneof" json:"RNG,omitempty"`
}

func (x *DynamicDataRequest) Reset() {...}

func (x *DynamicDataRequest) String() string {...}

...

func (x *DynamicDataRequest) GetCID() string {...}

func (x *DynamicDataRequest) GetLat() float64 {...}
```

**Listing 6.2**: `protoc` generated file `v1.dynamic.ais.pb.go`.

```

func (x *DynamicDataRequest) GetLon() float64 {...}

func (x *DynamicDataRequest) GetRNG() float64 {...}

...

type DynamicData struct {...}

func (x *DynamicData) Reset() {...}

...
```

**Listing 6.3**: `protoc` generated file `v1.dynamic.ais_grpc.pb.go`.

```
// Code generated by protoc-gen-go-grpc. DO NOT EDIT.
// versions:
// - protoc-gen-go-grpc v1.2.0
// - protoc             v3.21.5
// source: ais/v1.dynamic.ais.proto

package ais

import ...

// This is a compile-time assertion to ensure that this
// generated file is compatible with the grpc package it
// is being compiled against.
// Requires gRPC-Go v1.32.0 or later.
const _ = grpc.SupportPackageIsVersion7


// DynamicClient is the client API for Dynamic service.
// ...
type DynamicClient interface {...}

...

// DynamicServer is the server API for Dynamic service.
// All implementations must embed
// UnimplementedDynamicServer for forward compatibility
type DynamicServer interface {
    GetDynamics(
        *DynamicDataRequest,
        Dynamic_GetDynamicsServer
    ) error
    SubscribeDynamics(
        *DynamicDataRequest,
        Dynamic_SubscribeDynamicsServer
    ) error
    ExchangeDynamics(
        Dynamic_ExchangeDynamicsServer
    ) error
    ...
    mustEmbedUnimplementedDynamicServer()
}

// UnimplementedDynamicServer must be embedded to have
// forward compatible implementations.
type UnimplementedDynamicServer struct {
}

func (UnimplementedDynamicServer) GetDynamics(
    *DynamicDataRequest,
    Dynamic_GetDynamicsServer
) error {...}
func (UnimplementedDynamicServer) SubscribeDynamics(
    *DynamicDataRequest,
    Dynamic_SubscribeDynamicsServer
) error {...}
func (UnimplementedDynamicServer) ExchangeDynamics(
    Dynamic_ExchangeDynamicsServer
) error {...}
...
func (UnimplementedDynamicServer)
  mustEmbedUnimplementedDynamicServer() {}

...
```

The code-generation tool `protoc` generates all the necessary `interfaces`, `structs`, and `funcs` for both the gRPC server and client APIs. However, since only the server-side of the system is being built in Go, the client-side-related code can be ignored. To implement the interfaces, the generated code needs to be used. Listings 6.4 and 6.5 demonstrate the implementation of these interfaces.

**Listing 6.4**: Implementation of `DynamicServer` with Go.

```
package servers

import ...

type DynamicServer struct {
  ais.UnimplementedDynamicServer
}

func (s *DynamicServer) GetDynamics(
    req *ais.DynamicDataRequest,
    res ais.Dynamic_GetDynamicsServer
) error {...}

func (s *DynamicServer) SubscribeDynamics(
    req *ais.DynamicDataRequest,
    res ais.Dynamic_SubscribeDynamicsServer
) error {...}

func (s *DynamicServer) ExchangeDynamics(
    stream dynamic.Dynamic_DynamicQueryServer
) error {...}

...
```

**Listing 6.5**: Implementation of `main` function with Go.

```
package main

import ...

func main {

  listener, err := net.Listen("tcp", ":9090")

```

```
    ...

    grpcServer := grpc.NewServer()

    ais.RegisterStaticServer(
        grpcServer,
        &servers.StaticServer{}
    )
    ais.RegisterDynamicServer(
        grpcServer,
        &servers.DynamicServer{}
    )

    ...

    go grpcServer.Serve(listener)
    defer grpcServer.Stop()

    ...

}
```

Listing 6.4 provides the implementation of the `DynamicServer` interface, which is an essential part of the gRPC server. In Listing 6.5, the `main` function creates a new `grpcServer`, registers the services implemented in Listing 6.4, and listens for incoming messages from the client on port `9090`. This completes the construction of the gRPC server, and it is now ready to handle incoming requests from the client.

## 6.2 Implementing gRPC Client

In Chapter 4, three applications, BlueNavi, RedNavi, and GreenNavi, were designed and developed based on web technologies. However, web applications have certain limitations when it comes to supporting HTTP/2 and the gRPC communication framework. To address this, this section explores alternative frontend implementations that can work seamlessly with gRPC.

### 6.2.1 Background and Scenarios

The scenario presented in Section 4.1 involves small vessels that are not required to have onboard AIS equipment by IMO regulations or national laws. To improve navigation safety and vessel management, a cost-effective Class-B AIS equipment can be used alongside the BlueNavi system. This solution enables the OOW to capture information and movements of other vessels in the surrounding sea.

However, there is a prerequisite for this solution: the vessel must already be equipped with other devices or sensors, such as Global Positioning System (GPS), as AIS does not *produce* any data, and all data must be input from other sources. While small cargo ships often choose to equip themselves with GPS for positioning, this is not an option for smaller vessels such as fishing boats or yachts. Without a data source, AIS cannot transmit any information related to the vessel's position, speed, *etc.*, although this does not affect the reception of AIS messages from other vessels within the communication range.

Ensuring the safety of navigation is a mutual responsibility. Although large vessels are easy for smaller vessels to identify, small vessels may be more difficult for larger vessels to detect visually or by radar due to their smaller size. Therefore, it is important for both large and small vessels to have access to each other's navigational data to ensure safe navigation.

Today, many handheld devices are equipped with positioning solutions, in addition to dedicated GPS devices. Some newer devices even have dual-frequency GPS in the L1 band at 1575.42 MHz and L5 at 1176 MHz, which significantly increases GPS performance and positioning accuracy to the centimeter level. For example, in addition to dual-frequency GPS, the latest iPhone supports Global Navigation Satellite System (GLONASS), Galileo, Quasi-Zenith Satellite System (QZSS), and BeiDou. This has greatly increased the availability of mobile positioning. The portability, low cost, and high positioning accuracy of these devices make it possible to easier and more efficient to exchange navigational and other maritime information.

## 6.2.2 Tooling and Framework

Web applications can be run on mobile browsers by following responsive design principles[1], as demonstrated in Figure 6.1. However, since web applications run within the browser, their performance can be impacted by system restrictions on browser permissions for hardware like cameras and GPS, as well as software such as calendars and phone functions. As a result, the user experience and functionality of web applications are often limited.

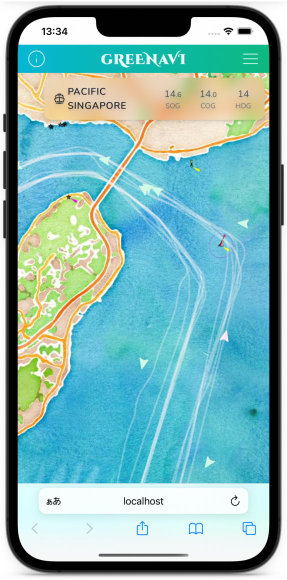


Figure 6.1: GreenNavi web application.

Native applications offer better performance and hardware access than web applications on mobile devices, but they can be costly to develop and maintain due to different development environments and Software Development Kits (SDKs) across platforms, resulting in increased investment when developing and maintaining two completely different applications in different technology stacks for different platforms, such as Swift or Objective-C for iOS and Kotlin or Java for Android.

In an effort to address the issue of development time and cost, *hybrid applications* have been proposed, which essentially wrap a native application shell around a web application to provide access to hardware resources and device APIs. While this shortens the development cycle and saves costs, the hybrid solution still relies on web application principles and falls short in terms of speed and user experience when compared to native applications.

A new approach, known as *cross-platform application* development, has emerged to tackle this challenge. This approach involves utilizing a cross-platform framework and a single set of code to develop applications for multiple platforms, providing the advantages of native applications while solving the problem of code reuse.

[1] Responsive design is a web design approach that aims to create a user interface that adapts to different devices and screen sizes, in order to optimize the user experience.

Among the various cross-application platforms available, such as Xamarin, Ionic, and React Native, this dissertation chose Flutter, which was released by Google in 2017, because of the following advantages.

**Near-to-Native Performance** Many other cross-platform frameworks utilize a *bridge* between the application and the platform, which maps the application's components, animations, *etc.* using JavaScript or other languages to native components on the platform to achieve functionality. However, this communication can adversely impact performance. In contrast, Flutter renders the UI directly on the canvas, resulting in near-native performance.

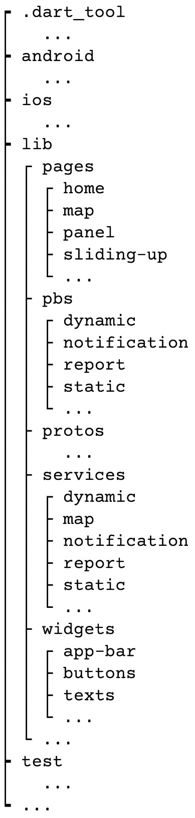
**UI Consistency** Because other cross-platform solutions rely on native components, the definition and implementation of these components often vary from platform to platform, leading to consistency issues. On the other hand, Flutter uses its rendering engine to render UI and animations directly to the canvas, reducing consistency issues.

**The Dart Language** Flutter uses the Dart language, which was also developed by Google. Dart has several features, including support for Just-in-Time (JIT) and ahead-of-time compilation, allowing for hot reloading during development without sacrificing efficiency in production environments.

Therefore, despite its relatively recent release, this dissertation selected Flutter as the client-side development language.

### 6.2.3 Implementation

The cross-platform mobile application, named *PinkNavi*, also adopts the MVC design pattern. The project structure of PinkNavi is illustrated in Figure 6.2, which provides an overview of the organization of the various application components.



```
.dart_tool
  ...
android
  ...
ios
  ...
lib
  pages
    home
    map
    panel
    sliding-up
    ...
  pbs
    dynamic
    notification
    report
    static
    ...
  protos
    ...
  services
    dynamic
    map
    notification
    report
    static
    ...
  widgets
    app-bar
    buttons
    texts
    ...
  ...
test
  ...
...
```

**Figure 6.2**: PinkNavi project structure.

The gRPC communication-related components are the focus of this subsection. Leveraging gRPC's native support for Dart used in Flutter, the `protoc` tool is again used to generate the necessary code for the client-side communication, as presented in Listing 6.6.

**Listing 6.6**: Command for generating code in Dart.

```
protoc --proto_path=proto \
--dart_out=grpc:./pbs protos/ais/v1.dynamic.ais.proto
```

In Listing 6.6, the `protoc` command generates four files: `v1.dynamic.ais.pb.dart`, `v1.dynamic.ais.pbenum.dart`, `v1.dynamic.ais.pbgrpc.dart`, and `v1.dynamic.ais.pbjson.dart`. These files contain message data structures, communication method implementations, and other necessary components. To use these generated files in the corresponding services, they need to be imported into the project. An example of how to call these methods is shown in Listing 6.7.

**Listing 6.7**: Implementation of `DynamicService` in Dart.

```
import ...

class DynamicService {

  Location location = Location();
  final String _clientID = "...";

  ...

  static DynamicClient stub = DynamicClient(
    ClientChannel(
      "...",
      port: 9090,
      options: ...,
    ),
  );

  LocationService() {
    _init();
  }

  Future<void> _init() async {...}

  Stream<DynamicData> getDynamics(...) async* {...}

  ...

  Stream<DynamicData> ExchangeDynamics(int range) async* {
    if (_allReady) {
      StreamController<DynamicDataRequest> streamController =
      StreamController<DynamicDataRequest>();
      _locationSubscription = location.onLocationChanged
          .listen((LocationData locationData) {
            streamController.add(DynamicDataRequest()
              ..cID = _clientID
              ..lat = locationData.latitude!
              ..lon = locationData.longitude!
              ..rNG = range
          );
      });
      final dynamics =
      stub.exchangeDynamics(streamController.stream);
      await for (DynamicData dynamic in dynamics) {
        yield dynamic;
      }
    }
  }

  ...

}
```

In the `DynamicService` class, the gRPC stub—`DynamicClient`—is first created using the code generated by `protoc`, and then the client-side business logic of the three gRPC methods is implemented. Once the functions are implemented, Flutter UI components (widgets) can use the gRPC service by calling the functions in the DynamicService class.

Figure 6.3 shows the final prototype of the PinkNavi cross-platform application (Android version followed by iOS version). By calling the platform API, the application retrieves the current location, speed, heading, and other relevant data about the own ship, and includes this information in the query request when calling the gRPC method to retrieve the dynamic information of the surrounding ships from the server-side. Furthermore, PinkNavi can calculate the target ships' bearing, distance, Distance to Closest Point of Approach (DCPA), Time to Closest Point of Approach (TCPA), *etc.*, based on the received data, as shown in Figure 6.4.

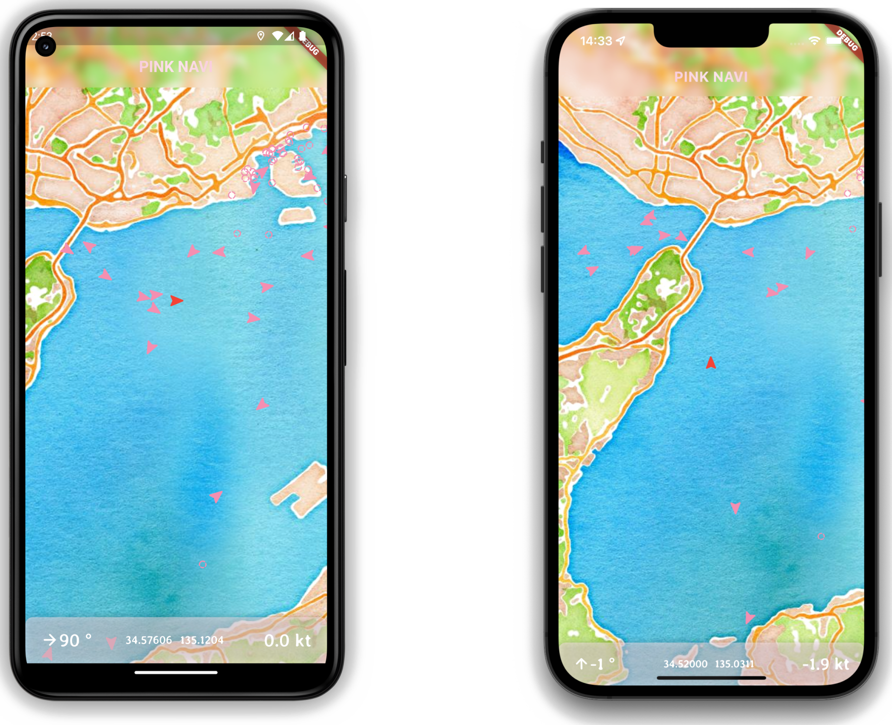
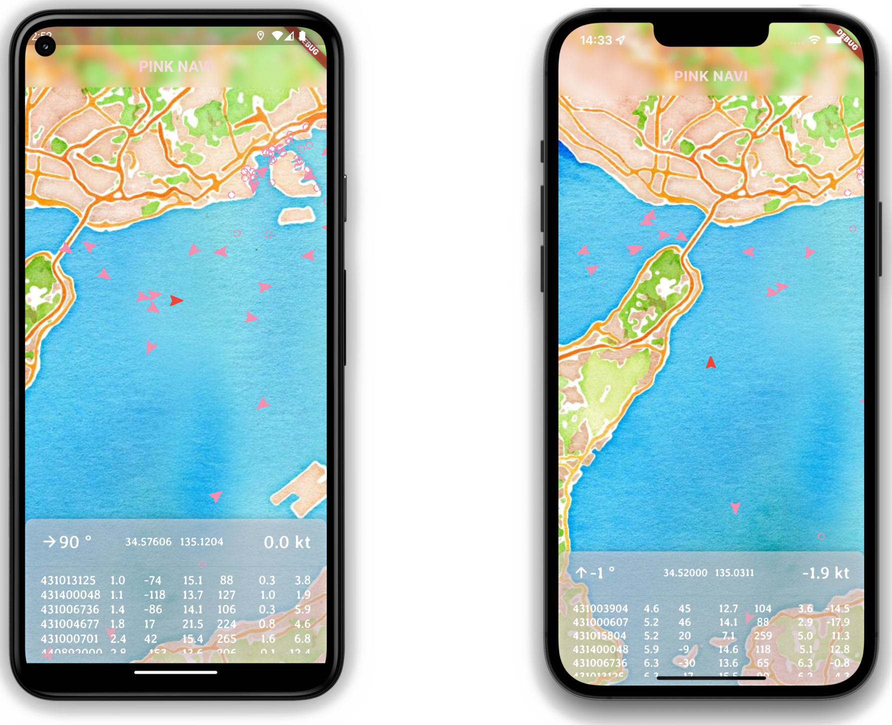


**Figure 6.3:** PinkNavi UI.



```
→90 °        34.57606  135.1204        0.0 kt

431013125  1.0   -74    15.1   88    0.3    3.8
431400048  1.1   -118   13.7   127   1.0    1.9
431006736  1.4   -86    14.1   106   0.3    5.9
431004677  1.8   17     21.5   224   0.8    4.6
431000701  2.4   42     15.4   265   1.6    6.8
440892000  2.8   -153   13.6   206   0.1   -12.4
```

```
↑-1 °        34.52000  135.0311        -1.9 kt

431003904  4.6   45     12.7   104   3.6   -14.5
431000607  5.2   46     14.1   88    2.9   -17.9
431015804  5.2   20     7.1    259   5.0    11.3
431400048  5.9   -9     14.6   118   5.1    12.8
431006736  6.3   -30    13.6   65    6.3   -0.8
431013125  6.3   -17    15.5   90    6.2    4.3
```

**Figure 6.4:** PinkNavi UI: sliding-up panel.

## 6.3 Summary

In Chapter 6, the implementation of gRPC services was discussed in detail, covering both the server-side and the client-side perspectives. The implementation of the gRPC server focused on performance and efficiency, while the implementation of the gRPC client explored alternatives for frontend development other than web applications, specifically cross-platform applications. With gRPC's code generation capabilities, remote communication between microservices or between clients and servers can be achieved simply by calling remote services as local functions.

# Modern Deployment Approach and Microservices Containerization

# Deploying Microservices 7

In Chapter 2, it was mentioned that microservices architecture involves breaking down services into relatively individual and independent microservices, based on functional or other principles. Consequently, unlike a monolithic system that only needs to be deployed only on a single server, microservices need to be deployed separately. Furthermore, even the same microservice may need to be deployed in different environments if the requirements are different. As a result, *containerization* has been proposed as an idea to simplify the tedious deployment work. This chapter delves into the deployment of microservices using this containerization technology.

## 7.1 Deploying Microservices on Docker

In 2013, *Docker* introduced a technology called containerization, which has since gained widespread adoption. This section explores the concept of containerization, the benefits it provides, and how microservices can be deployed using Docker.

### 7.1.1 Packing Services to Containers

In modern operating systems, the *kernel* manages hardware resources and any process that needs to access these resources must communicate with the kernel via a *system call*. The kernel then directs the request to the appropriate resource, as depicted in Figure 7.1.

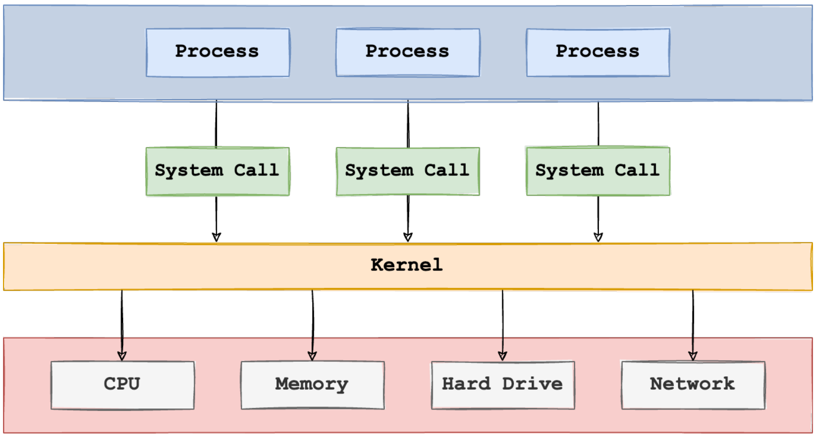


**Figure 7.1:** Concept of operating system.

Ensuring consistent and stable system environments across development, testing, and production stages can be challenging due to differences in hardware, software configurations, and dependencies. Even if a service passes testing in the development environment, environmental differences in the production environment can cause various problems.

To address this challenge, *virtual machines* on different hardware have been used to unify the runtime environment. As illustrated in Figure 7.2, a virtual machine runs on top of the host operating system and communicates with the host kernel through a hypervisor. Each virtual machine has its own isolated operating system environment where different processes interact only with the virtual machine's operating system. This way, as long as the same virtual machine is configured, services can run in the same virtual environment, regardless of the underlying host operating system.

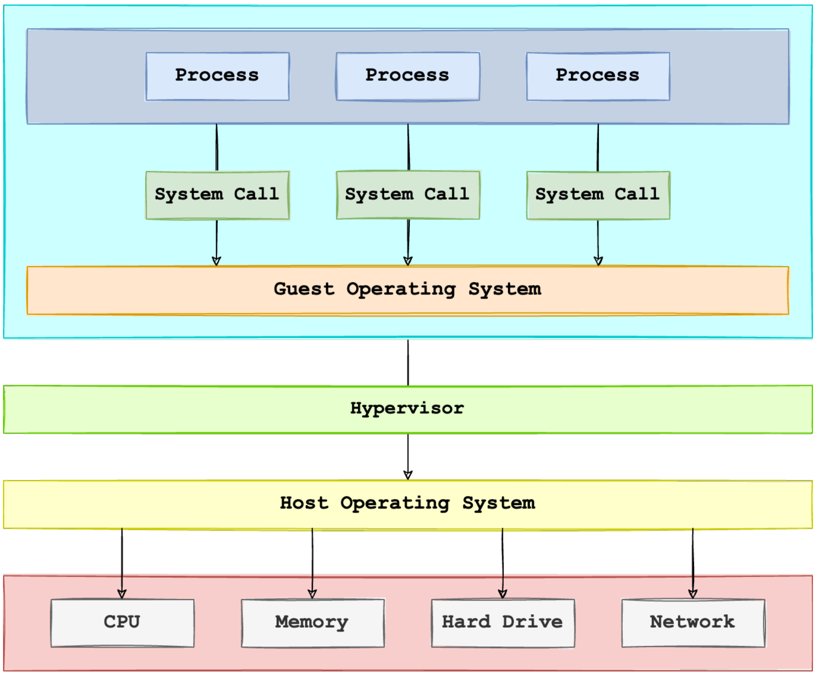


**Figure 7.2:** Concept of virtual machine.

However, virtual machines are often considered too cumbersome, resource-intensive and slow to boot, making them less than ideal for deploying systems at scale. This is especially true for microservices, where each microservice may require an isolated runtime environment. Starting and running multiple virtual machines simultaneously consumes a significant amount of resources. As a result, containerization has emerged as an alternative solution.

*Containers* can be thought of as lightweight virtual machines. However, unlike virtual machines, multiple containers can run on the same host and share the same system kernel[1], as shown in Figure 7.3. The operating system can allocate different portions of hardware resources to different containers without interfering with each other, which allows the container to be considered an isolated runtime environment.

In Docker, a container is essentially an instance of an *image*. An image contains a snapshot of the files needed to run the container and the command to start the container, as shown in Figure 7.4. When a container is created, Docker copies the snapshot from the image to the container's hard drive and starts the container using the command specified in the image. Once the container is started, it runs independently.

Images, on the other hand, are built from Dockerfiles. A Dockerfile is a script that defines the steps needed to create an image. For example, Listing 7.1 shows the Dockerfile for building an image of a App Provider microservice (for BlueNavi) developed in Go, while Listing 7.2 provides the command to build the image using the Dockerfile.

[1] Different containers are essentially independent processes that can be isolated from each other thanks to the *namespaces* and *control groups* features of the Linux system.

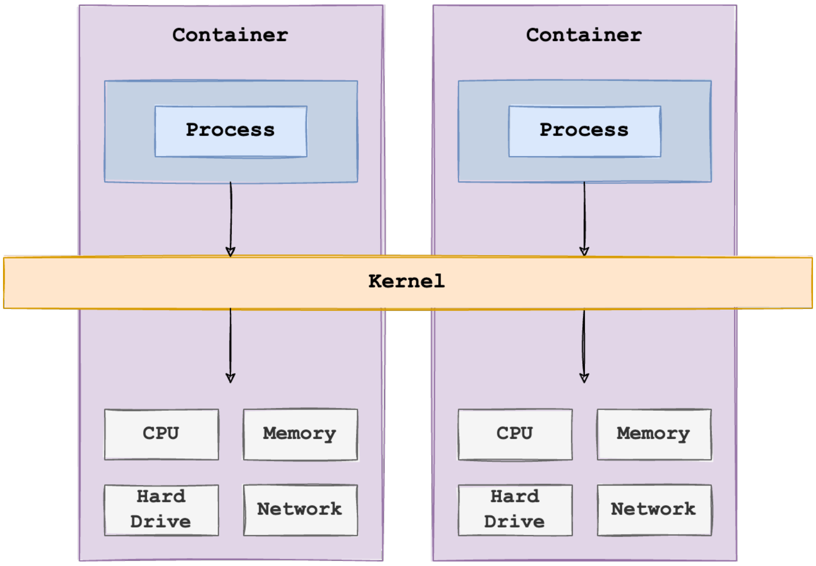


**Figure 7.3:** Concept of container.

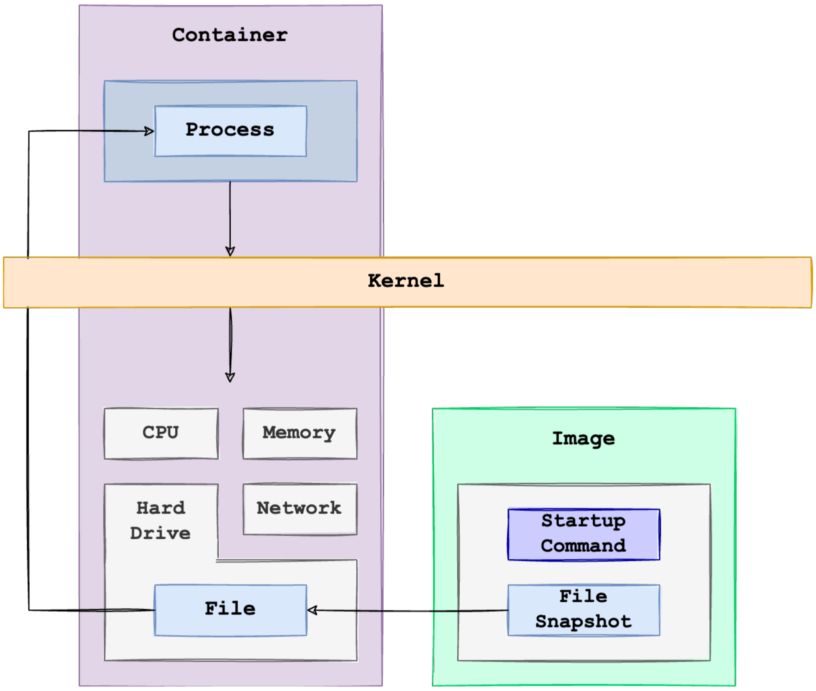


**Figure 7.4:** Container built from image.

**Listing 7.1:** `Dockerfile` for building microservice App Provider.

```
FROM golang:1.18-alpine as builder

WORKDIR /go/src/blue-server

COPY ./blue-server ./
RUN CGO_ENABLED=0 GOOS=linux go build -o /blue-server

FROM gcr.io/distroless/base-debian11

LABEL maintainer = "..."

WORKDIR /

COPY --from=builder /blue-server /blue-server
COPY --from=builder /go/src/blue-server/app/dist /app/dist

USER nonroot:nonroot
ENTRYPOINT ["/blue-server"]
```

**Listing 7.2**: Command for building the docker image.

```
docker build -t .../blue-server:latest .
```

The Dockerfile in Listing 7.1 demonstrates the creation of a lightweight container image for a Go server named `blue-server`, containing only the necessary files to run the application. The build process consists of two phases. In the first phase, an image named `builder` is built using the `golang:1.18-alpine` image as the base. The Go code for the web application is copied to the working directory `/go/src/blue-server`, and the binary is built and copied to the root directory. In the second phase, the final image is built using the `gcr.io/distroless/base-debian11` image. The working directory is set to `/`, and the binary built in the first phase is copied from the builder image to the `/app/dist` directory, which contains the distribution package or assets required for running the BlueNavi application. Finally, the entry point is set to the binary, allowing the container to run independently.

Containerization enables consistent deployment of applications in production environments and also facilitates deployment on platforms with different architectures. For example, while most computers currently in use have Complex Instruction Set Computers (CISC) architectures such as x86 and ARM64, using a miniature and affordable computer like Raspberry Pi, as depicted in Figure 7.5, to deploy microservices can significantly reduce costs. However, Raspberry Pi is a Reduced Instruction Set Computer (RISC) that uses the ARMv7 architecture, making cross-platform compilation and deployment a challenging task. Fortunately, Docker simplifies this task. For instance, Listing 7.3 outlines the commands to build an ARMv7 platform image `blue-server-armv7:latest` on an ARM64 architecture platform using the `docker buildx` command.

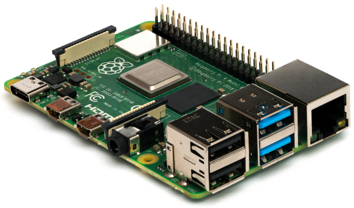


**Figure 7.5**: Raspberry Pi [58].

**Listing 7.3**: Commands for building an ARMv7 image on a different platform.

```
docker buildx create --name xbuilder
docker buildx use xbuilder
docker buildx inspect --bootstrap
docker buildx build --platform linux/arm/v7 \
-t blue-server-armv7:latest \
-o type=docker,dest=- . > blue-server-armv7.tar
```

### 7.1.2 Going Live with Services

After the container image is built, it can be used to start a container with the `docker run` command. To make the container accessible from outside the host, the `-p` flag is used to expose the container's internal ports, as demonstrated in Listing 7.4.

**Listing 7.4**: Command for starting the docker container.

```
docker run -p 80:9900 .../blue-server:latest .
```

When deploying the `blue-server`, port `9900` is used to provide the services. To make the service accessible from a browser using a domain name or IP address, the `-p` flag is used with the docker run command to expose the container's internal port `9900` to port `80` on the host, as demonstrated in Listing 7.4.

However, the BlueNavi application provided by `blue-server` needs to interact with the backend microservices to retrieve data. Because they are independent, the backend microservices should be run in separate and isolated containers. Although the microservices can be launched using Docker commands, manually configuring the network and other settings can be a tedious and error-prone task. Instead, *docker-compose* can be used to simplify the process.

Docker-compose is a separate Command-Line Interface (CLI) tool that is installed with Docker and is used to launch multiple Docker containers simultaneously. It automates some of the verbose and time-consuming arguments of the `docker run` command. The docker-compose configuration is defined in a YAML file, which is shown in Listing 7.5.

```
version: '3'
services:
  ais-receiver:
    build:
      context: ./ais-receiver
    ports:
      - '52000:3000'
    restart: unless-stopped
  ais-provider:
    build:
      context: ./ais-provider
    ports:
      - '9000:9000'
    restart: unless-stopped
  blue-server:
    build:
      context: ./blue-server
    ports:
      - '80:9900'
    restart: unless-stopped
  ...
  mongodb:
    image: 'mongo:6.0'
    volumes:
      - './mongodb/data:/data/db'
    restart: always
```

**Listing 7.5:** `docker-compose.yaml` for configuring docker-compose.

Listing 7.5 configures four Docker containers: `ais-receiver`, `ais-provider`, `blue-server`, and `mongodb`. Each container's image file is created from the Dockerfile located in the specified path. The `ais-receiver` container needs to receive forwarded AIS messages via the User Datagram Protocol (UDP) port, so it must be exposed to the corresponding external port. The `ais-provider` container listens on port `9000` to provide AIS data, while the app-provider container must be directly accessible via the browser and thus requires exposure to external port `80`. Since containers are better suited for stateless services, a Docker volume is used to map the data stored on the host to the `mongodb` container to ensure data persistence.

Once the configuration is complete, the command in Listing 7.6 can be used to build and launch all microservices simultaneously.

```
docker-compose up --build
```

**Listing 7.6:** Command for starting microservices using `docker-compose`.

## 7.2 Deploying Microservices on Kubernetes

The Docker technology allows microservices to be containerized and deployed on the host server. As described in Chapter 2, microservice architectures offer many advantages over traditional monolithic architectures, such as increased flexibility, availability, and scalability. However, the benefits of scalability and availability may not be fully realized in Docker deployments since they often extend beyond the capabilities of containers. To address this, this section will delve into the deployment of microservices through *container orchestration* technologies, specifically the *Kubernetes* platform, to attain high scalability and availability.

### 7.2.1 Orchestrating Containers with Kubernetes

In the context of large-scale systems, ensuring high availability and scalability is essential. While containerization offers several benefits, such as flexibility and portability, it may not be sufficient to address the aforementioned concerns on its own. This is where container orchestration technologies come into play. Kubernetes, an open-source platform developed and released by Google, is a leading choice for container orchestration. Its popularity has been growing rapidly and it has become the *de facto* industry standard.

Kubernetes provides a rich set of features that enable automatic scaling, rolling updates, and self-healing capabilities for containerized applications. Figure 7.6 illustrates a simplified model of the Kubernetes architecture, which consists of a cluster of *nodes* that host the containers and a set of control plane components that manage and orchestrate the containers.

As illustrated in Figure 7.6, Kubernetes operates by orchestrating containers in *clusters*. Each cluster consists of nodes, which are either physical or virtual hosts capable of running containers. Inside each node are *pods*, which are the smallest units of Kubernetes that manage containers. A pod can run a single container or a set of containers that work together to implement a specific business logic. Containers within the same pod share the same resources and network, making communication between them fast and efficient. The *master component* manages all nodes in the cluster and is responsible for orchestrating the containers. Figure 7.7 provides a higher-level view of the Kubernetes cluster architecture.

### 7.2.2 Going Live with Services

Kubernetes offers two methods for deploying systems: declaratively or imperatively. This subsection will deploy the system declaratively, using configuration files.

The system configuration diagram for deployment on the Kubernetes cluster is shown in Figure 7.8. All deployment configuration files in this section are based on this diagram.

Figure 7.8 introduces several Kubernetes *abstractions*, including *Deployment*, *Service*, *PVC*, and *Ingress*. These abstractions provide higher-level

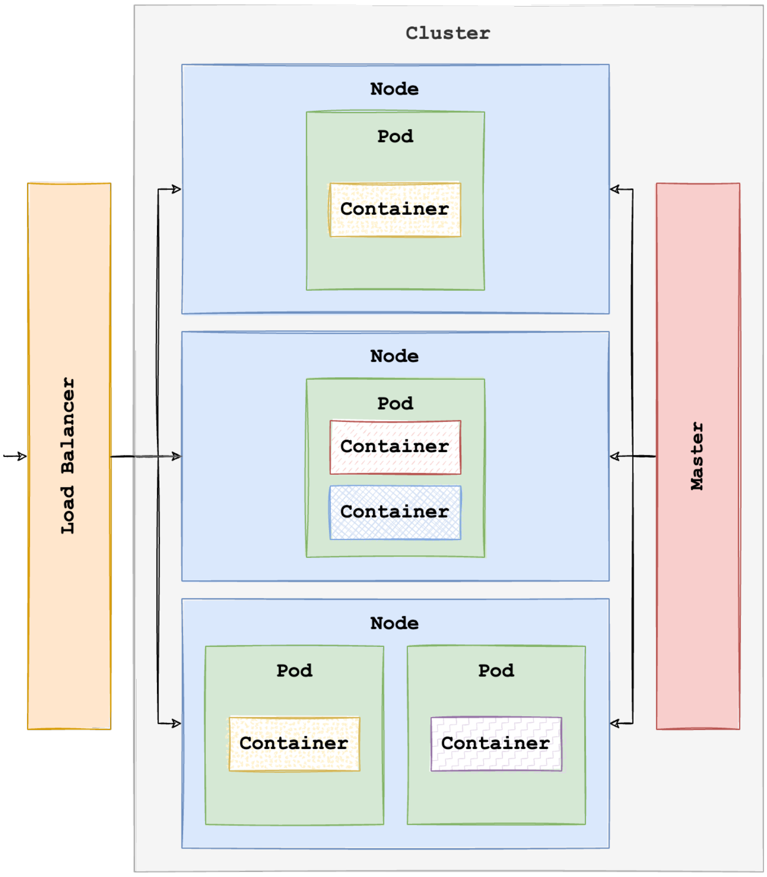


**Figure 7.6:** Concept of Kubernetes.

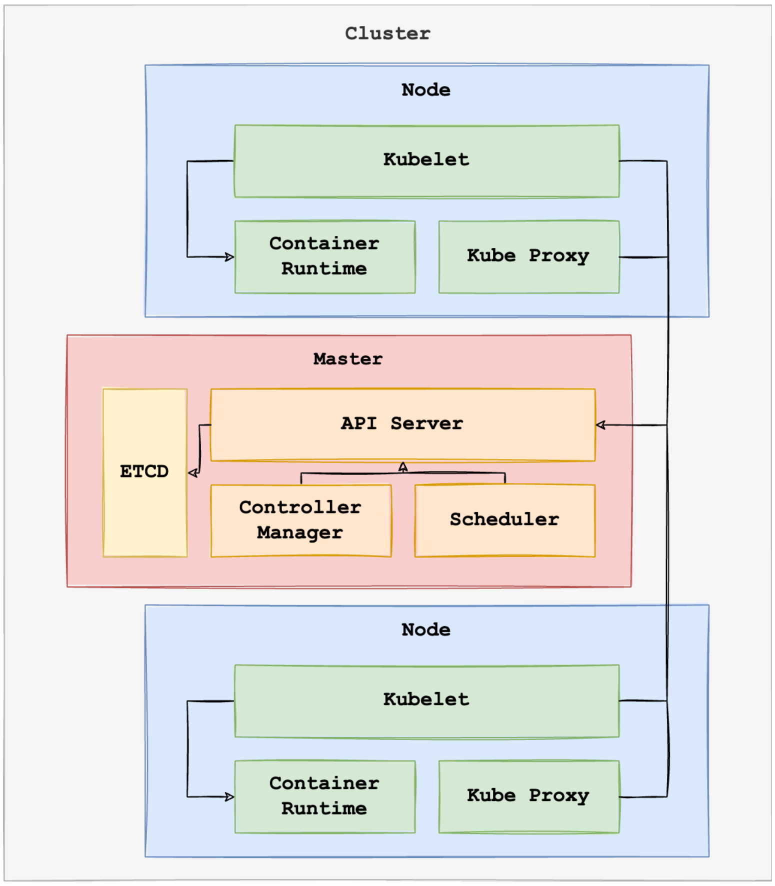


**Figure 7.7:** Kubernetes cluster.

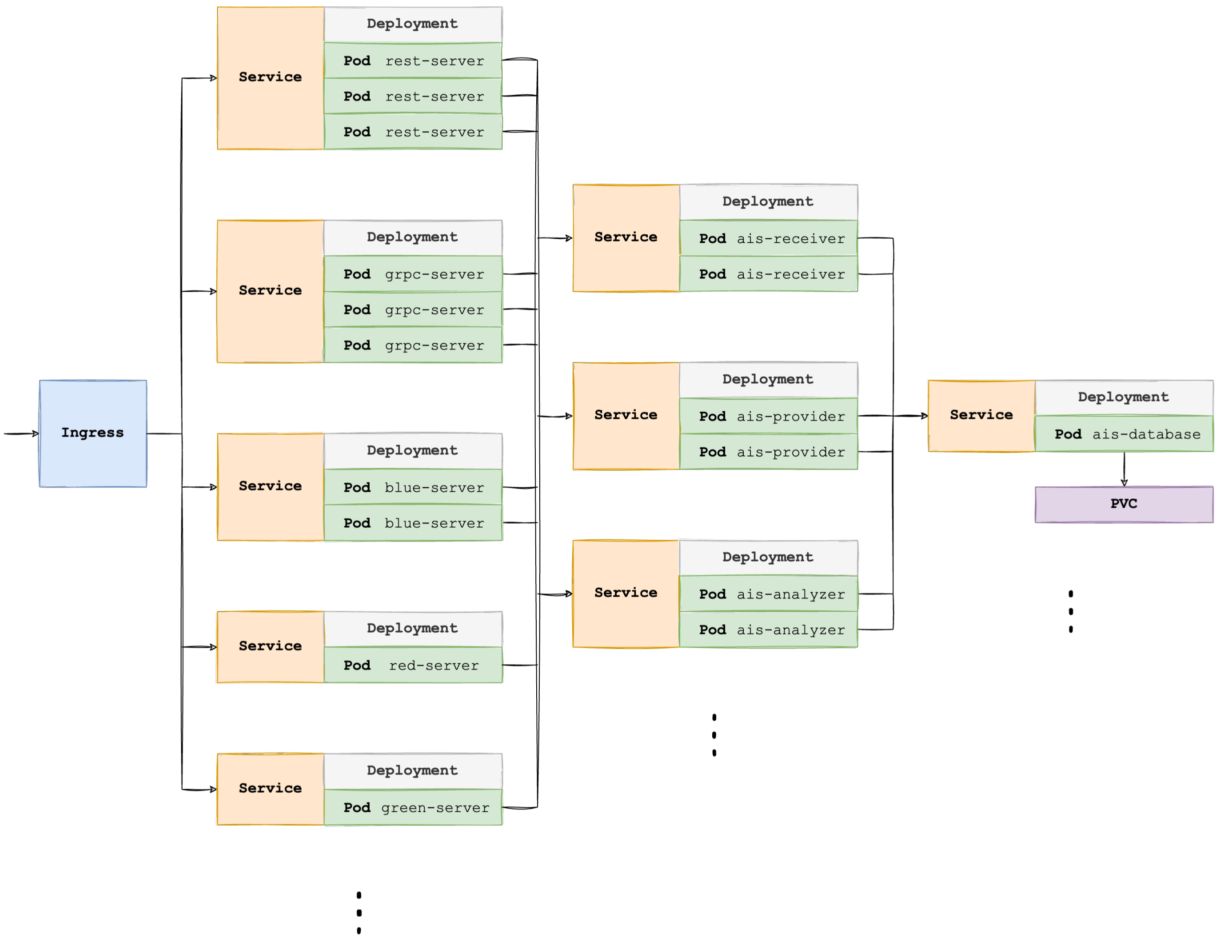


**Figure 7.8:** Deploying microservices with Kubernetes.

views of the system and make it easier to manage and deploy the components. To complete the declarative deployment of the entire system, different configuration files must be created for each abstraction.

### Configuring Deployments

As explained in Section 7.2.1, Kubernetes manages containers through the Pod abstraction, but it does so indirectly through the Deployment abstraction. The Deployment manages the scaling and updating of Pods, allowing declarative management of the desired state of each microservice, including the desired number of replicas. Listing 7.7 shows the YAML file for the Deployment configuration of the `grpc-server` microservice depicted in Figure 7.8.

**Listing 7.7:** `grpc-server-deployment.yaml` for configuring `gRPCServer`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grpc-server-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      component: grpc-server
  template:
    metadata:
      labels:
        component: grpc-server
    spec:
      containers:
        - name: grpc-server
          image: .../grpc-server
          ports:
            - containerPort: 9090
              name: grpc-port
```

### Configuring Services

The Service abstraction in Kubernetes facilitates communication between pods within a cluster. Services ensure that there are consistent endpoints for pods, even if their underlying IPs change. Services also enable communication between the pods and external clients. The YAML Service configuration file for the `grpc-server` is given in Listing 7.8. By specifying the service type, the port, and the selector to match the label of the pods, the Service is created to expose the `grpc-server` pods.

**Listing 7.8**: `grpc-server-service.yaml` for configuring `gRPCServer`.

```
apiVersion: v1
kind: Service
metadata:
  name: grpc-server-service
spec:
  type: LoadBalancer
  selector:
    component: grpc-server
  ports:
    - port: 80
      targetPort: 9090
      name: grpc-port
```

### Configuring PVC

Containers are better suited for *stateless services*. For *stateful services*, such as a database, a better approach is to put the data in a Persistent Volume (PV) that can store data permanently and persist even if the Pod fails. The Persistent Volume Claim (PVC) is a Kubernetes abstraction used to declare the storage requirements of the Pod. In other words, it specifies how much storage is required for the Pod to function properly. Listing 7.9 provides the YAML file for configuring a PVC.

**Listing 7.9**: `ais-database-pvc.yaml` for configuring a PVC.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ais-database-pvc
spec:
  accessModes:
    - ReadWriteOnce
resources:
  requests:
    storage: 10Gi
```

### Configuring Ingress

In Kubernetes, Ingress acts as a reverse proxy that routes external traffic to the appropriate Services within the cluster. It provides an easy and flexible way to manage external access to microservices. Ingress is placed in front of the Services and listens for requests coming from external sources. Based on the hostname and path of the request, Ingress routes the requests to the appropriate Service. Listing 7.10 provides a YAML file for configuring Ingress and enabling external access to the gRPC server.

**Listing 7.10**: `ingress.yaml` for configuring Ingress.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    cert-manager.io/cluster-issuer: letsencrypt-prod
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
  - hosts:
    - ...
    - ...
    secretName: ...-tls
  rules:
  - host: ...
    http:
      paths:
      - path: /...
        pathType: Prefix
        pathRewrite: /...
        backend:
          service:
            name: grpc-server-service
            port:
              name: grpc-port
  - ...
```

### Configuring TLS

Ensuring secure access to the system is an essential aspect that must be taken into account in designing and developing a modern maritime information support system. In order to protect the transmission of data over the internet, the HTTPS protocol is usually employed, with encryption provided by Transport Layer Security (TLS) or Secure Sockets Layer (SSL). In Kubernetes, a TLS certificate can be configured to secure communication between clients and services. Listings 7.11 and 7.12 provide the YAML configuration files for two different abstractions, *ClusterIssuer* and *Certificate*, respectively. The ClusterIssuer is used to specify the issuer of the certificate, while the Certificate is used to define the Domain Name System (DNS) names and the *Secret* abstraction to use for encryption. The Secret contains the TLS certificate and key, which are used to encrypt the data exchanged between clients and servers. By configuring TLS, sensitive information can be protected from unauthorized access and eavesdropping, thereby enhancing the overall security of the system.

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: ...
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
    - http01:
        ingress:
          class: nginx
```

**Listing 7.11**: `issuer.yaml` for configuring the TLS certificate issuer.

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: ...-tls
spec:
  secretName: ...-tls
  issuerRef:
    name: letsencrypt-prod
  dnsNames:
  - ...
  - ...
```

**Listing 7.12**: `certificate.yaml` for configuring the TLS certificate service.

## 7.3 Summary

Chapter 7 discussed the deployment of modern maritime information support system microservices in a containerized environment. The use of containerization technology was proposed to simplify the deployment process, which is otherwise cumbersome due to the granularization of microservices into individual, independent components. Both Docker and Kubernetes provide efficient and effective methods for deploying microservices, with Docker offering a straightforward deployment solution and Kubernetes offering high scalability and availability capabilities, which are essential for demonstrating the benefits of a microservices architecture. Overall, this chapter highlighted the important role of containerization and container orchestration in deploying microservices and the benefits they provide in terms of flexibility, availability, and scalability.

# Testing Microservices 8

The previous chapters defined and implemented both REST and gRPC-based microservices for serving AIS data. This chapter will test these services by deploying them to a remote server.

## 8.1 General Idea and Test Environment

This chapter aims to compare the performance of the REST API and the gRPC API developed in the previous chapters. To achieve this, two specific tasks will be tested: querying static AIS information for a single vessel and querying dynamic AIS information for multiple vessels in the surrounding sea. These tasks were chosen for the following reasons:

- Single-vessel static AIS data has a small data size, while multi-vessel dynamic AIS data has a large data size. Comparing the performance of the REST and gRPC APIs in handling different data payloads can provide insight into their relative strengths and weaknesses;
- The gRPC API uses unary RPC for single-ship AIS static data queries, while the REST API uses standard REST communication. Comparing the performance of these two communication modes can evaluate the effectiveness of using gRPC's unary RPC for static navigational information queries;
- The gRPC API uses server-streaming RPC for multi-ship AIS dynamic data queries, while the REST API uses standard REST communication. Comparing the performance of these two communication modes can evaluate the effectiveness of using gRPC's server-streaming RPC for dynamic navigational information queries.

To test the performance of the two APIs, three microservices have been developed using different communication models and programming languages. These services are identified as follows:

**`grpc-go`** Microservice built in Go and provides gRPC services.
**`rest-go`** Microservice built in Go and provides REST services.
**`rest-js`** Microservice built in Node.js and provides REST services.

The three services were built based on the microservices defined in the previous chapters, with some modifications made for testing purposes. These modifications were implemented for the following reasons:

- To ensure consistent data for each test, the response returned by each service is static and represents a fixed point in time, rather than real-time data from the AIS database. This is achieved by terminating the data reception process, *i. e.*, the service is not actively receiving data and the database is not being updated;

- To avoid any performance issues caused by the cold start of the service and data transfer between the database and the service, all data is read into memory in advance to simulate the actual operation of the service;
- All computational tasks have been removed from the workflow of the services, with the services solely focused on returning the queried data;
- The tests have been streamlined for simplicity, excluding other features such as user authentication. This also means that the size of the request header and body (if applicable) is smaller than what would be needed in a real production environment.

The microservices used for testing are deployed in a containerized fashion, as described in Chapter 7, on the Google Kubernetes Engine (GKE)[1] using the `n2d-standard-2` machine type. The test environment has two second-generation AMD EPYC series CPUs clocked at 2.25 GHz, 8.34 GB of memory, and 29.34 GB of ephemeral storage. However, since the Kubernetes system requires certain resources, the specific resources that can be allocated to each service are slightly smaller, as shown in Table 8.1.

[1] A managed service for running containerized applications on Google Cloud Platform (GCP).

**Table 8.1:** Overview of test environment.

| Service Label | API Type | Language | Allocatable CPU | Allocatable Memory | Allocatable Storage |
|---|---|---|---|---|---|
| `grpc-go` | gRPC | Go | 1.93 CPU | 6.33 GB | 9.23 GB |
| `rest-go` | REST | Go | 1.93 CPU | 6.33 GB | 9.23 GB |
| `rest-js` | REST | Node.js | 1.93 CPU | 6.33 GB | 9.23 GB |

All tests are performed on a physical machine equipped with a 10-core Apple M1 Pro chip, 16 GB of LPDDR5 memory with a maximum bandwidth of 200 GB/s, and an APPLE SSD. The operating system used is macOS Monterey version 12.5.1.

## 8.2 Test Methods and Test Results

To test the performance of the microservices, intensive requests are sent from the client side to the services remotely deployed on GKE (server side) for a duration of 60 seconds, and the responses are recorded. Statistics are collected both on the client and server sides, and the analysis is based on these statistics. The performance test is conducted with the following specific parameters:

**Test Duration** 60 seconds;
**gRPC Client Concurrency** 1 or 10 (number of gRPC request workers);
**gRPC Client Connections** 1 or 10 (concurrency is evenly distributed across the connections, *i. e.*, one connection per concurrency);
**REST Client Concurrency** 1 or 10 (number of REST request workers);
**Requests per Second (QPS)** 0 (no limit on the request rate);
**gRPC Request Body Payload** 0 to 5 bytes;
**REST Request Body Payload** not applicable to the HTTP `GET` method.

To ensure the reliability and generality of the test results, performance testing was conducted at nine different locations, including:

- Nemuro-shi, Hokkaido, Japan;
- Otaru-shi, Hokkaido, Japan;
- Hakodate-shi, Hokkaido, Japan;
- Akita-shi, Akita, Japan;
- Osaka-shi, Osaka, Japan;
- Izumo-shi, Shimane, Japan;
- Hiroshima-shi, Hiroshima, Japan;
- Shimonoseki-shi, Yamaguchi, Japan;
- Sasebo-shi, Nagasaki, Japan.

Testing in multiple locations ensures that the results are not specific to a single geographic area and better captures the real-world performance of the APIs in different scenarios. The average of the tests at these locations was then used for analysis. Specifically, the following items were compared and analyzed across services:

Client-side statistics:

**Request Body Payload** Size of the payload for each request (in bytes);
**Response Body Payload** Size of the payload for each response (in bytes);
**Throughput** Number of successful data transfers between server and client over the test duration.
**Average Latency** Average of all the latency values (in milliseconds);
**Latency Distribution** Distribution of the latency values (in milliseconds);
**Status Distribution** Distribution of the status for each response (in percent).

Server-side statistics:

**Image Size** Size of each containerized service image (in megabytes);
**CPU Usage** Number of CPUs used by each service;
**Memory Usage** Memory used by each service (in megabytes).

The test results are summarized in Table 8.2.

**Table 8.2:** Summary of test result.

| Task | Single-Ship Static Data | | | | | | Multi-Ship Dynamic Data | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Service | `grpc-go` | | `rest-go` | | `rest-js` | | `grpc-go` | | `rest-go` | | `rest-js` | | |
| Concurrency | 1 | 10 | 1 | 10 | 1 | 10 | 1 | 10 | 1 | 10 | 1 | 10 | |
| Request Body | 5 | | – | | – | | 0 | | – | | – | | B |
| Response Body | 99 | | 203 | | 203 | | 8400 | | 16619 | | 16619 | | B |
| Throughput | 2311 | 22979 | 1140 | 8264 | 1122 | 8164 | 1844 | 12109 | 750 | 4095 | 738 | 4108 | # |
| Latency / Avg. | 30.7 | 29.6 | 61.8 | 75.3 | 62.4 | 77.8 | 38.4 | 138.5 | 95.8 | 200.5 | 97.8 | 236.5 | ms |
| Latency / 50% | 27.3 | 26.9 | 57.8 | 59.3 | 58.9 | 59.7 | 34.8 | 109.3 | 89.3 | 156.8 | 88.8 | 155.4 | ms |
| Latency / 75% | 29.7 | 29.7 | 62.2 | 66.7 | 63.2 | 67.6 | 39.8 | 163.9 | 98.6 | 223.8 | 98.2 | 253.7 | ms |
| Latency / 90% | 34.9 | 35.6 | 68.9 | 83.2 | 70.0 | 84.4 | 48.1 | 239.1 | 112.3 | 332.7 | 117.4 | 475.2 | ms |
| Latency / 95% | 40.6 | 42.2 | 76.2 | 101.0 | 76.3 | 99.7 | 62.2 | 301.2 | 121.4 | 429.7 | 133.8 | 592.1 | ms |
| Latency / 99% | 71.4 | 72.9 | 118.0 | 335.8 | 128.0 | 378.1 | 95.8 | 528.1 | 167.0 | 784.5 | 189.1 | 1067.3 | ms |
| Status OK | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | % |
| Image Size | 35.2 | | 35.77 | | 125.4 | | 35.2 | | 35.77 | | 125.4 | | MB |
| CPU Usage | 0.010 | 0.055 | 0.004 | 0.025 | 0.013 | 0.051 | 0.029 | 0.085 | 0.020 | 0.027 | 0.040 | 0.052 | # |
| Memory Usage | 8.67 | 10.14 | 9.28 | 9.56 | 43.39 | 50.99 | 9.23 | 10.66 | 9.71 | 9.64 | 43.28 | 45.59 | MB |

## 8.3 Discussion

The results in Table 8.2 show that the payloads of gRPC are smaller than those of REST due to the compact binary format of the protocol buffers, compared to the more verbose text-based format of JSON commonly used in REST. Additionally, gRPC's built-in support for streaming and flow control can reduce payload sizes in certain scenarios. For instance, when a client requests static data for a ship of MMSI 235050802, the associated services return the result, where the message body size of the REST API is approximately 203 bytes, while the gRPC API with protocol buffers transmits a message body of only 99 bytes.

In addition, gRPC's use of HTTP/2 as its transport protocol enables multiplexing, which allows for multiple requests and responses to be transmitted simultaneously over a single connection, reducing the overhead associated with opening and closing multiple connections. This feature not only increases server throughput and reduces data transfer latency, but also proves to be particularly advantageous when using server-streaming RPC for dynamic data transfers, as demonstrated in Table 8.2.

While gRPC offers benefits such as smaller payloads and multiplexing over HTTP/2, it also requires binary encoding and decoding of messages, which can make the service more computationally expensive compared to its REST counterpart. In addition, the flow control and multiplexing features of HTTP/2 may introduce additional overhead. As shown in the table, the gRPC service may consume more CPU resources than the REST service.

To mitigate the increased CPU usage of gRPC, it is critical to choose the appropriate hardware and optimize the service implementation. With careful hardware selection, efficient algorithms, and effective caching strategies, the benefits of gRPC can outweigh the costs, resulting in higher performance and better scalability.

The choice of programming language can have a significant impact on the performance of the API services. As shown in the table, Go has advantages over Node.js in terms of image size, CPU usage, and memory usage, while still providing similar performance metrics like throughput and latency. This is because Go has a more lightweight standard library, produces statically linked binaries, and uses a more efficient memory management system. In contrast, Node.js includes its entire runtime environment in the image and has a less efficient garbage collection mechanism, which can lead to higher CPU and memory usage.

However, it is important to note that there is no one-size-fits-all solution when it comes to choosing a programming language for developing web APIs. The choice of language depends on various factors, including project requirements, existing technology stack, and development team expertise. Therefore, this dissertation does not make any definitive conclusions about which programming language is superior; it simply evaluates the performance differences between Go and Node.js in specific scenarios.

Ultimately, when designing and developing a modern maritime information support system, developers should carefully consider the project requirements, evaluate different programming languages based on their performance, and choose the language that best meets the needs of the industry.

## 8.4 Summary

Chapter 8 evaluated the performance of the REST and gRPC APIs for serving AIS data. Three services were created for testing and the results showed that gRPC had smaller payloads and better server throughput due to its use of the compact binary format protocol buffers and HTTP/2 as the transport protocol. However, gRPC had higher CPU usage due to binary encoding and decoding. The comparison between Go and Node.js showed that microservices developed in Go have a smaller image size, lower CPU usage, and more efficient memory management. In practice, however, the choice of programming language depends on various factors. Overall, this chapter highlighted the importance of carefully evaluating the performance of different APIs and programming languages when designing, developing, and deploying modern maritime information support systems. In particular, the gRPC framework and the Go programming language are preferred when the right conditions are met, as they provide excellent performance benefits and efficiency.

# Conclusions and Outlook 9

This chapter provides insights into the key findings and outcomes of the research conducted, as well as an outlook on future research directions and potential areas for improvement in the field.

## 9.1 Conclusions

In this dissertation, a comprehensive approach was proposed for designing, developing, and deploying modern maritime information support systems. The approach emphasizes the use of flexible software architectures, efficient communication mechanisms, and versatile application forms to address the complexities of modern maritime information support systems.

Through thorough analysis and discussion, it was concluded that microservices architecture is a suitable solution for systems with numerous functions and unevenly distributed loads. The proposed approach granularizes the monolithic system into microservices to reduce the coupling between services, leading to improved scalability, availability, and maintainability of the system.

It should be noted that while microservices architecture may not be the best solution for every system, the proposed approach provides a flexible and adaptable framework for addressing the challenges faced by modern maritime information support systems. The approach offers a potential solution for organizations looking to modernize their systems, while also providing a foundation for further research and development in the field.

In systems with a microservices architecture, communication between services is a critical aspect to consider. Different communication methods, such as synchronous or asynchronous, can be used, each with its own advantages and disadvantages. Synchronous communication, such as using RESTful APIs, provides reliable message transmission, but may have a negative impact on efficiency. On the other hand, asynchronous communication, such as message broadcasting, can improve efficiency by eliminating acknowledgement steps, but may sacrifice reliability. To ensure effective communication in microservice systems, it is important to carefully evaluate the specific requirements of the system and choose the appropriate communication mechanism accordingly.

The role of the backend system in this approach is to handle data and primary and heavy business logic, while the frontend is responsible for presenting views and interacting with users. The two systems communicate through APIs, with RESTful API being the most widely adopted solution. However, gRPC, a newer approach released in 2015, offers benefits such as improved performance, consistency, and stability, making it a more suitable option for communication between microservices. Nevertheless, gRPC's lack of browser-side compatibility makes it inappropriate for web applications, which is why this dissertation adopted both RESTful and gRPC APIs as the primary means of inter-service communication.

For the frontend, compatibility is a critical factor to ensure smooth usage of the relevant services on a variety of devices, regardless of platform architecture or operating system. Three options were considered to ensure compatibility: web applications, hybrid applications, and cross-platform applications. Of these options, web applications offer the most compatibility as they can run on almost all modern browsers using the latest frontend frameworks or libraries. However, the limitations of browsers at the operating system level and the performance of the browsers themselves can often result in a less-than-optimal user experience and limited access to device hardware and software resources. To balance compatibility, resource accessibility, and user experience at a lower cost, this dissertation chose to use cross-platform applications in certain application scenarios.

Containerization technology was introduced to address the problem of developing, testing, and deploying applications across heterogeneous environments, including processors with different architectures. Services were containerized using Docker, making them independent, portable, and reusable. For simple deployment tasks, such as deploying a system on a ship LAN, microservices could be deployed using Docker to achieve process isolation. However, once the scope of services, scale of data, and complexity of the system reached a certain level, container orchestration technologies were considered necessary for deploying microservices at scale. In the industry, Kubernetes had become the *de facto* standard for container orchestration.

The BlueNavi frontend application was designed and developed to address practical issues in the maritime industry. Using a portable Class-B AIS device as the data source, the backend microservices can be deployed in a portable single-board computer and connected to the frontend through the ship's LAN. This solution effectively tackles the problem of AIS non-carrying vessels being unable to receive real-time navigational information from their surroundings. Additionally, the system demonstrates its flexibility by allowing the backend deployment to be omitted if the user has internet access and can instead utilize cloud-based backend services.

In the field of MET, RedNavi was designed and developed to make 1D or 2D navigational information three-dimensional. By constructing a computer-generated 3D representation of the navigational information, RedNavi serves as a useful tool for education, training, or even for assisting OOWs with their lookouts. Furthermore, RedNavi and BlueNavi can leverage the same backend services and data (*i. e.*, the same API), enabling users who need both systems to avoid redundant deployments. This demonstrates the versatility and efficiency of the system.

Finally, the cross-platform application, PinkNavi, was designed and developed to bring the full potential of hardware resources and performance of user devices, especially smartphones with built-in GPS. As a result, small fishing boats and yachts without navigational equipment such as AIS and GPS are now able to exchange maritime information, especially navigational information. Furthermore, various communication patterns have been created to cater to different application scenarios and requirements, enabling further exploration into the integration of maritime information exchange based on IP communication.

## 9.2 Outlook

The purpose of this dissertation is to thoroughly investigate the design, development, and deployment of modern maritime information support systems. However, given the constraints of time, resources, and individual abilities, not all aspects of the topic could be explored in depth. This section will highlight some of the issues that were not fully addressed or under-explored in the previous chapters of the dissertation, with the aim of providing insights for future research.

### Distributed Data and Database

In Chapter 7, this dissertation investigated the implementation of Kubernetes container orchestration technology to ensure the reliability and scalability of the system's services. However, the data are stored in a PV, which may not be sufficient for large-scale systems. There are several challenges to consider when dealing with data storage and management in these systems, including:

**Scalability** As the system grows and the number of users increases, the amount of data and Input/Output (I/O) load can become enormous, potentially exceeding the processing capacity of a single server host. In these cases, it is necessary to consider distributing the data or load.

**Availability** Inevitable failures of a single data server can affect the functionality of the entire system. To mitigate this, redundancy or backups from other data centers are necessary.

**Latency** The geographic proximity of data and services to users can have a significant impact on system latency and user experience. In particular, time-sensitive data such as AIS dynamic information should be stored on the nearest data server to minimize access latency.

**Data Consistency** Keeping data consistent and up-to-date across all data servers can be a challenge in a distributed database system.

**Security** Protecting the sensitive and critical data stored in the system is paramount. Therefore, it is necessary to consider security measures such as encryption, access control, and regular backups.

To address these challenges, a variety of technologies and techniques can be used, including sharding, clustering, partitioning, and data replication. In addition, it is necessary to evaluate and select appropriate database management systems that can support the needs of the system, including distributed data management, high-performance data processing, and real-time data analysis. However, implementing these techniques requires careful planning and design considerations.

### Security

Security is an important aspect of any system, but it is underrepresented in the research conducted in this dissertation. The backend APIs, developed in Chapters 3 and 6, were built using REST and gRPC, respectively, and were exposed to the frontend applications and third parties in Chapter 7. This, however, poses a security risk as exposing more portals to the public increases the system's vulnerability to attack.

In a production environment, security must be a top priority, and it must be considered from two perspectives: communication channel security and authentication and access control. Although in Section 7.2.1, the use of HTTPS and TLS certificates for encrypting communications over the internet was briefly mentioned, the dissertation does not delve into the topic of user authentication, which is crucial for identifying visitors and their service requests.

Moreover, measures must be taken to prevent unauthorized access and to protect sensitive data from malicious attacks, such as Man-in-The-Middle Attacks (MITMs), SQL injection, and Cross-Site Scripting (XSS).

In summary, security should be treated as a critical factor in the development and deployment of modern maritime information support systems, and there is a need for further research to address these security concerns in depth.

### Network Reliability

Despite the reliable communication mechanism provided by TCP and IP at the transport layer, data transmission still faces physical limitations. For instance, early satellite communications for internet access had significant latency issues, which only improved with advances in satellite communication technology.

Similarly, data exchange solutions that rely on 4G / 5G technology also face coverage and signal strength limitations, particularly at sea. To ensure optimal service coverage and utilization, it is necessary to make improvements to the existing communication infrastructure from a technology standpoint.

### User Experience

The design and development of maritime information support systems should take into account the critical aspect of user experience. The frontend web applications or cross-platform applications designed in Chapters 4 and 6 are only prototypes studied at the system architecture level. When these applications become products, it is essential that user experience be considered in their design and development, from the interaction logic of the entire system to the design of each component of the user interface. This includes ease of use, user-centered design, intuitive navigation, and visually appealing graphics.

Human-Computer Interaction (HCI) is a theoretical domain that covers user experience design. However, practical considerations are just as important in ensuring the success and adoption of these applications in the marketplace. The user experience can be improved through continuous user testing and feedback, as well as by staying abreast of the latest HCI design trends and best practices. This helps to meet the evolving needs and preferences of users and provide a seamless and enjoyable experience. Therefore, further research is needed to explore and optimize the user experience in these systems, from design to testing and feedback.

### Non-Technical Factors

In addition to technical advancements, the success of modern maritime information support systems is also significantly influenced by various non-technical factors. These include culture, policy, laws and regulations, as well as the revision of international conventions that govern maritime activities worldwide.

The establishment of industry standards and their widespread implementation is a critical challenge facing the industry. This requires the combined efforts and collaboration of various stakeholders, including technical and non-technical experts, industry players, government agencies, and international organizations. A multidisciplinary approach is necessary to address this complex issue and ensure the harmonization of industry standards and practices.

In conclusion, non-technical factors are just as important as technical ones in shaping the development and success of modern maritime information support systems. Further research is needed to identify the non-technical challenges facing the industry, and to find ways to address them effectively to ensure its growth and success.

# Bibliography

Here are the references in citation order.

[1] PETER BELLWOOD. *MAN'S CONQUEST OF THE PACIFIC - THE PREHISTORY OF SOUTHEAST ASIA AND OCEANIA.* New York: Oxford University Press, Dec. 31, 1978 (cited on page 1).

[2] Stan Lusby, Robert Hannah, and Peter Knight. "Navigation and Discovery in the Polynesian Oceanic Empire." In: *Hydrographic Journal* 131.132 (2009), pp. 17–25 (cited on page 1).

[3] United Nations Conference on Trade and Development. *Review of Maritime Transport 2022.* United Nations, Geneva, Nov. 2022, p. 195 (cited on page 1).

[4] Goran Dominioni and Dominik Englert. *Carbon Revenues From International Shipping: Enabling an Effective and Equitable Energy Transition - Technical Paper.* Technical Paper. Washington, DC: World Bank, Apr. 1, 2022. (Visited on 02/11/2023) (cited on page 1).

[5] H. Gatty. *The Raft Book: Lore of the Sea and Sky.* G. Grady Press, 1943 (cited on page 1).

[6] Great Britain Navy Department. *Admiralty Manual of Navigation: BR 45(1).* BR Series 1. Stationery Office, 1997 (cited on page 1).

[7] R. Jones. *The Lighthouse Encyclopedia: The Definitive Reference.* Second edition. G - Reference, Information and Interdisciplinary Subjects Series. Globe Pequot Press, 2004 (cited on page 1).

[8] United States Hydrographic Office. *The International Code of Signals: For the Use of All Nations.* Amer., 1890 (cited on page 1).

[9] Carlota Perez. "Technological revolutions and techno-economic paradigms." In: *Cambridge Journal of Economics* 34.1 (Sept. 2009). _eprint: https://academic.oup.com/cje/article-pdf/34/1/185/4756731/bep051.pdf, pp. 185–202. doi: `10.1093/cje/bep051` (cited on page 2).

[10] Marija Jović et al. "Digitalization in Maritime Transport and Seaports: Bibliometric, Content and Thematic Analysis." In: *Journal of Marine Science and Engineering* 10.4 (Apr. 2022). Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, p. 486. doi: `10.3390/jmse10040486`. (Visited on 02/12/2023) (cited on page 2).

[11] *Digitalization in the maritime industry.* DNV. Mar. 8, 2021. url: `https://www.dnv.com/Default` (visited on 02/12/2023) (cited on page 2).

[12] John Maguire. *The History of Maritime Communication and More!* CruiseDirect.com. Mar. 15, 2019. url: `https://www.cruisedirect.com/the_history_of_maritime_communication` (visited on 12/04/2022) (cited on page 2).

[13] MIC. *Maritime Communications.* MIC The Radio Use Website | License | Maritime Communications. July 17, 2015. url: `https://www.tele.soumu.go.jp/e/adm/system/satellit/marine/` (visited on 12/04/2022) (cited on page 2).

[14] Statista Research Department. *Market share of mobile telecommunication technologies worldwide from 2016 to 2025, by generation.* Aug. 11, 2022 (cited on page 3).

[15] ST Engineering iDirect. "IP Trunking." Dec. 4, 2020. (Visited on 12/04/2022) (cited on page 3).

[16] IALA. *G1117: VHF Data Exchange System (VDES) overview (Edition 2.0).* Dec. 15, 2017 (cited on pages 3, 5).

[17] Alena Kabelov¿ and Libor Dost¿lek. *Understanding TCP/IP: A clear and comprehensive guide to TCP/IP protocols.* Illustrated edition. Birmingham: Packt Publishing, May 11, 2006. 480 pp. (cited on page 4).

[18] IMO. *Resolution A.1106(29) – Revised Guidelines for The Onboard Operational Use of Shipborne Automatic Identification Systems (AIS).* Dec. 2, 2015. url: `https://wwwcdn.imo.org/localresources/en/OurWork/Safety/Documents/AIS/Resolution%20A.1106(29).pdf` (cited on pages 4, 35, 43).

[19] IHO. *Navigation Warnings on the Web*. Navigation Warnings on the Web | IHO. Dec. 21, 2021. URL: https://iho.int/navigation-warnings-on-the-web (visited on 11/30/2022) (cited on page 4).

[20] Pero Vidan, Josip Kasum, and Marijan Zujić. "Meteorological Navigation and ECDIS." In: *PROMET - Traffic&Transportation* 22 (Sept. 2010). DOI: 10.7307/ptt.v22i5.202 (cited on page 4).

[21] IMO. *International Convention for the Safety of Life at Sea (SOLAS), 1974*. Nov. 1, 1974. URL: https://www.imo.org/en/About/Conventions/Pages/International-Convention-for-the-Safety-of-Life-at-Sea-(SOLAS),-1974.aspx (visited on 11/06/2022) (cited on pages 4, 35).

[22] IMO. *Resolution A.851(20) – General Principles for Ship Reporting Systems and Ship Reporting Requirements*. Nov. 27, 1997. URL: https://wwwcdn.imo.org/localresources/en/OurWork/Safety/Documents/AIS/Resolution%20A.1106(29).pdf (cited on page 4).

[23] IMO. "Shipping Emergencies - Search and Rescue and the GMDSS." In: (Mar. 1999) (cited on page 4).

[24] Colin R. Pratt and Geoff Taylor. "AIS – A Pilot's Perspective." In: *Journal of Navigation* 57.2 (2004). Publisher: Cambridge University Press, pp. 181–188. DOI: 10.1017/S0373463304002772 (cited on page 5).

[25] Marko Perkovic et al. "The use of integrated maritime simulation for education in real time." In: *ResearchGate/Pub* 228912986 (2013), pp. 461–478 (cited on page 5).

[26] Dong Yang et al. "How big data enriches maritime research – a critical review of Automatic Identification System (AIS) data applications." In: *Transport Reviews* 39.6 (2019), pp. 755–773. DOI: https://doi.org/10.1080/01441647.2019.1649315 (cited on page 5).

[27] Joakim Trygg Mansson, Margareta Lutzhoft, and Ben Brooks. "Joint Activity in the Maritime Traffic System: Perceptions of Ship Masters, Maritime Pilots, Tug Masters, and Vessel Traffic Service Operators." In: *Journal of Navigation* 70.3 (2017). Publisher: Cambridge University Press, pp. 547–560. DOI: 10.1017/S0373463316000758 (cited on page 5).

[28] Bernhard Berking. "Potential and benefits of AIS to Ships and Maritime Administrations." In: *WMU Journal of Maritime Affairs* 2.1 (Apr. 1, 2003), pp. 61–78. DOI: 10.1007/BF03195034. (Visited on 12/04/2022) (cited on page 5).

[29] Centre of Excellence for Operations in Confined and Shallow Waters et al. "From Fragmented Sea Surveillance to Coordinated Maritime Situational Awareness." In: *Maritime Situational Awareness* (Apr. 2015) (cited on page 5).

[30] Eberhard Wolff. *Microservices: Flexible Software Architecture*. Google-Books-ID: zucwDQAAQBAJ. Addison-Wesley Professional, Oct. 3, 2016. 861 pp. (cited on pages 13, 14).

[31] Thomas Erl. *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. 2nd. USA: Prentice Hall Press, 2016. 416 pp. (cited on page 13).

[32] Arne Koschel, Irina Astrova, and Jeremias Dötterl. "Making the move to microservice architecture." In: *2017 International Conference on Information Society (i-Society)*. 2017 International Conference on Information Society (i-Society). July 2017, pp. 74–79. DOI: 10.23919/i-Society.2017.8354675 (cited on page 13).

[33] Davi Monteiro et al. "Building orchestrated microservice systems using declarative business processes." In: *Service Oriented Computing and Applications* 14.4 (Dec. 1, 2020), pp. 243–268. DOI: 10.1007/s11761-020-00300-2. (Visited on 10/28/2022) (cited on page 14).

[34] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. 2nd Edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly Media, Aug. 10, 2021. 400 pp. (cited on page 14).

[35] Kasun Indrasiri and Prabath Siriwardena. *Microservices for the Enterprise: Designing, Developing, and Deploying*. 1st Edition. New York, NY: Apress, Nov. 15, 2018. 444 pp. (cited on pages 17, 18).

[36] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. 1st edition. Beijing Köln: O'Reilly Media, Oct. 5, 2010. 448 pp. (cited on page 18).

[37] ITU. *M.1371 : Technical characteristics for an automatic identification system using time division multiple access in the VHF maritime mobile frequency band*. 2014. URL: https://www.itu.int/rec/R-REC-M.1371-5-201402-I/en (visited on 10/24/2022) (cited on page 23).

[38] Urs Ramer. “An iterative procedure for the polygonal approximation of plane curves.” In: *Computer Graphics and Image Processing* 1.3 (Nov. 1, 1972), pp. 244–256. DOI: 10.1016/S0146-664X(72)80017-0. (Visited on 02/15/2023) (cited on page 30).

[39] David H Douglas and Thomas K Peucker. “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature.” In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 10.2 (Dec. 1973). Publisher: University of Toronto Press, pp. 112–122. DOI: 10.3138/FM57-6770-U75U-7727. (Visited on 02/15/2023) (cited on page 30).

[40] Christopher Ireland et al. “A classification of object-relational impedance mismatch.” In: First International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA). Cancun, Mexico, Mar. 2009. (Visited on 12/04/2022) (cited on page 31).

[41] Andy Norris. “AIS Implementation – Success or Failure?” In: *The Journal of Navigation* 60.1 (Jan. 2007). Publisher: Cambridge University Press, pp. 1–10. DOI: 10.1017/S0373463307004031. (Visited on 11/06/2022) (cited on page 35).

[42] awwwards.com. *What are Frameworks? 22 Best Responsive CSS Frameworks for Web Design.* Feb. 20, 2013. URL: https://www.awwwards.com/what-are-frameworks-22-best-responsive-css-frameworks-for-web-design.html (visited on 11/07/2022) (cited on page 37).

[43] Google. *Angular - Introduction to the Angular Docs.* 2022. URL: https://angular.io/docs (visited on 11/07/2022) (cited on page 38).

[44] Hongze Liu and Nobukazu Wakabayashi. “RedNavi: Building a 3D Scene of the Current Sea from AIS Data.” In: *Sustainability* 14.19 (2022). DOI: 10.3390/su141912572 (cited on page 40).

[45] Tao Liu, Depeng Zhao, and Mingyang Pan. “An approach to 3D model fusion in GIS systems and its application in a future ECDIS.” In: *Computers & Geosciences* 89 (Apr. 1, 2016), pp. 12–20. DOI: 10.1016/j.cageo.2016.01.008. (Visited on 04/05/2022) (cited on page 40).

[46] Yuzuru Isoda et al. “Reconstruction of Kyoto of the Edo era based on arts and historical documents: 3D urban model based on historical GIS data.” In: *International Journal of Humanities and Arts Computing* 3.1 (Oct. 1, 2009). Publisher: Edinburgh University Press, pp. 21–38. DOI: 10.3366/ijhac.2009.0007. (Visited on 04/22/2022) (cited on page 40).

[47] Sarah Batson, Robert Score, and Alex J. Sutton. “Three-dimensional evidence network plot system: covariate imbalances and effects in network meta-analysis explored using a new software tool.” In: *Journal of Clinical Epidemiology* 86 (June 2017), pp. 182–195. DOI: 10.1016/j.jclinepi.2017.03.008 (cited on page 40).

[48] R. Cwilewicz and L. Tomczak. “Improvement of ship operational safety as a result of the application of virtual reality engine room simulators.” In: *Risk Analysis Vi: Simulation and Hazard Mitigation.* Ed. by C. A. Brebbia. WOS:000256668100052. Southampton: Wit Press/Computational Mechanics Publications, 2008, pp. 535–544. DOI: 10.2495/RISK080521. (Visited on 04/25/2022) (cited on page 40).

[49] Evangelos Markopoulos et al. “Maritime Safety Fducation with VR Technology (MarSEVR).” In: *2019 10th Ieee International Conference on Cognitive Infocommunications (coginfocom 2019).* ISSN: 2375-1312 WOS:000582418600048. New York: Ieee, 2019, pp. 283–288. (Visited on 04/25/2022) (cited on page 40).

[50] Steven C. Mallam, Salman Nazir, and Sathiya Kumar Renganayagalu. “Rethinking Maritime Education, Training, and Operations in the Digital Era: Applications for Emerging Immersive Technologies.” In: *Journal of Marine Science and Engineering* 7.12 (Dec. 2019). Number: 12 Publisher: Multidisciplinary Digital Publishing Institute, p. 428. DOI: 10.3390/jmse7120428. (Visited on 04/25/2022) (cited on page 40).

[51] Roy Thomas Fielding. “Architectural Styles and the Design of Network-based Software Architectures.” PhD thesis. UNIVERSITY OF CALIFORNIA, IRVINE, 2000 (cited on pages 51, 52).

[52] Kasun Indrasiri and Danesh Kuruppu. *GRPC - Up and Running: Building Cloud Native Applications With Go and Java for Docker and Kubernetes.* Boston: Oreilly & Associates Inc, Feb. 11, 2020. 188 pp. (cited on pages 51, 54).

[53] *GRPC Core: g_stands_for.* Oct. 14, 2022. URL: https://grpc.github.io/grpc/core/md_doc_g_stands_for.html (visited on 11/13/2022) (cited on page 52).

[54] casey.crane. *The Advantages of HTTP2 – Why You Should Move on to HTTP2*. Cheap SSL Security. July 26, 2019. URL: https://cheapsslsecurity.com/p/the-advantages-of-http2/ (visited on 12/04/2022) (cited on page 53).

[55] Ilya Grigorik. *HTTP/2: A New Excerpt from High Performance Browser Networking*. O'Reilly, May 1, 2015 (cited on page 53).

[56] Roberto Peon and Herve Ruellan. *HPACK: Header Compression for HTTP/2*. Request for Comments RFC 7541. Num Pages: 55. Internet Engineering Task Force, May 2015. DOI: 10.17487/RFC7541. (Visited on 11/11/2022) (cited on page 54).

[57] Sufyan bin Uzayr. *GoLang: The Ultimate Guide*. CRC Press, Dec. 20, 2022 (cited on page 64).

[58] Laserlicht. *Deutsch: Raspberry Pi 4 Model B von der Seite*. July 3, 2019. (Visited on 02/18/2023) (cited on page 80).

# Special Terms

**Numbers**

**1D** One-Dimensional. 40, 96

**2.5G** Second and a Half Generation of Wireless Mobile Telecommunications Technology. 2

**2D** Two-Dimensional. 29, 36, 40, 41, 47, 96

**2G** Second Generation of Wireless Mobile Telecommunications Technology. 2

**3D** Three-Dimensional. 7, 40, 41, 47, 96

**3G** Third Generation of Wireless Mobile Telecommunications Technology. 2, 3

**4G** Fourth Generation of Cellular Communications Standards. 2, 3, 98

**5G** Fifth Generation of Cellular Mobile Communications. 2, 3, 98

A

**AIS** Automatic Identification System. 4, 7, 8, 15–18, 23–26, 28, 29, 31–33, 35, 36, 38, 40, 41, 43, 46, 47, 56, 60, 61, 68, 69, 81, 89, 93, 96, 97

**API** Application Programming Interface. 7–9, 14, 16, 18, 21, 35, 38, 40, 43, 44, 46, 51–53, 56, 64, 67, 69, 71, 89–93, 95, 96, 98

**ARM** Advanced RISC Machines. 80

**ASCII** American Standard Code for Information Interchange. 26

**ASM** Application Specific Message. 3

C

**CISC** Complex Instruction Set Computers. 80

**CLI** Command-Line Interface. 81

**COG** Course Over Ground. 23, 59

**CORBA** Common Object Request Broker Architecture. 51

**CPU** Central Processing Unit. 14, 90–93

**CSS** Cascading Style Sheets. 36, 37

D

**DCPA** Distance to Closest Point of Approach. 72

**DGNSS** Differential Global Navigation Satellite System. 24

**DIY** Do-It-Yourself. 35

**DNS** Domain Name System. 86

**DOM** Document Object Model. 38, 44, 45

**DSC** Digital Selective Calling. 4

E

**ECDIS** Electronic Chart Display and Information System. 35, 36

**EDGE** Enhanced Data Rates for GSM Evolution. 2

**EOS** End of Stream. 55

**EPIRB** Emergency Position-Indicating Radio Beacon. 4

**ETA** Estimated Time of Arrival. 23

G

**GCP** Google Cloud Platform. 90

**GKE** Google Kubernetes Engine. 90

**GLONASS** Global Navigation Satellite System. 69

**GMDSS** Global Maritime Distress and Safety System. 4

**GPRS** General Packet Radio Service. 2

**GPS** Global Positioning System. 68, 69, 96, 97

**GT** Gross Tonnage. 35

**H**

**HCI** Human-Computer Interaction. 98, 99

**HF** High Frequency. 2, 4

**HOL** Head-of-Line. 53, 54

**HTML** Hyper Text Markup Language. 36, 37, 44, 52

**HTS** High Throughput Satellite. 3

**HTTP** Hypertext Transfer Protocol. 18, 46, 51–54, 90

**HTTP/2** Version 2 of the Hypertext Transfer Protocol. 9, 51–54, 61, 68, 92, 93

**HTTPS** Hypertext Transfer Protocol Secure. 86, 98

**I**

**I/O** Input/Output. 97

**IALA** International Association of Marine Aids to Navigation and Lighthouse Authorities. 3, 4

**ID** Identifier. 23–26, 32, 57–59

**IEEE** Institute of Electrical and Electronics Engineers. 3, 58, 59

**IF** Intermediate Frequency. 2

**IMO** International Maritime Organization. 4, 7, 43, 58, 68

**IP** Internet Protocol. 4, 5, 7, 35, 36, 38, 80, 85, 97, 98

**J**

**JCG** Japan Coast Guard. 38

**JIT** Just-in-Time. 70

**JS** JavaScript. 36, 37, 40, 41, 44, 63, 64, 70

**JSON** JavaScript Object Notation. 52, 53, 92

**JST** Japan Standard Time. 44

**L**

**LAN** Local Area Networks. 16, 38, 96

**LPDDR** Low-Power Double Data Rate. 90

**LRIT** Long-Range Identification and Tracking. 4

**LTE** Long Term Evolution. 3

**M**

**MET** Maritime Education and Training. 7, 40, 96

**MF** Medium Frequency. 2

**MITM** Man-in-The-Middle Attack. 98

**MKD** Minimum Keyboard Display. 36

**MMSI** Maritime Mobile Service Identity. 24, 41, 52, 57–59, 92

**MVC** Model-View-Controller. 38, 70

**N**

**NAVTEX** Navigational Telex. 3, 4

**NMEA** National Marine Electronics Association. 25

**O**

**OOW** Officer of the Watch. 43, 68, 96

**OSI** Open Systems Interconnection. 4

**P**

**PV** Persistent Volume. 85, 97

**PVC** Persistent Volume Claim. 82, 85

**Q**

**QPS** Requests per Second. 90

**QZSS** Quasi-Zenith Satellite System. 69

**R**

**REST** Representational State Transfer. 8, 9, 17, 18, 21, 51–53, 55, 56, 63, 89, 90, 92, 93, 95, 96, 98

**RISC** Reduced Instruction Set Computer. 80

**RMI** Remote Method Invocation. 51

**ROT** Rate of Turn. 59

**RPC** Remote Procedure Call. 51–61, 63, 89, 92

**S**

**SAR** Search and Rescue. 24, 43

**SART** Search and Rescue Transponde. 4

**SDK** Software Development Kit. 69

**SOG** Speed Over Ground. 23, 59

**SOLAS** International Convention for the Safety of Life at Sea. 4, 35

**SQL** Structured Query Language. 32, 98

**SSD** Solid-State Drive. 90

**SSL** Secure Sockets Layer. 86

**T**

**TCP** Transmission Control Protocol. 4, 53, 54, 98

**TCPA** Time to Closest Point of Approach. 72

**TLS** Transport Layer Security. 86, 98

**TTL** Time To Live. 16, 32

**U**

**UDP** User Datagram Protocol. 81

**UI** User Interface. 16, 37, 39, 41–45, 47, 70–72

**URI** Uniform Resource Identifier. 18

**URL** Uniform Resource Locator. 18, 44

**UTC** Coordinated Universal Time. 24

**V**

**VDE** VHF Data Exchange. 3

**VDES** VHF Data Exchange System. 3, 4

**VDL** VHF Data Link. 4, 5

**VHF** Very-High Frequency. 2–4, 16

**W**

**WebGL** Web Graphics Library. 40, 41

**X**

**XML** Extensible Markup Language. 52, 53

**XSS** Cross-Site Scripting. 98

**Y**

**YAML** YAML Ain’t Markup Language. 52, 81, 84–86

# Acknowledgements

Thank you to my supervisor, Professor Wakabayashi Nobukazu, for your dedicated guidance and support throughout my dissertation journey. Your expertise in the field and your willingness to make time for me were invaluable. I am grateful for your trust in me and for allowing me to explore my topic of interest. Your generosity in supporting me in this endeavor and providing me with the resources and tools necessary to complete my dissertation was greatly appreciated. It has been a valuable learning experience.

Thank you to my co-authors, Professor Irena Jurdana and Assistant Professor Nikola Lopac, for your great contributions to the research presented in *BlueNavi: A Microservices Architecture-Styled Platform Providing Maritime Information*. Your expertise, insights, and hard work have been essential to the success of this work. I am grateful for the opportunity to have collaborated with you.

Thank you to my committee members, Professor Kohsaka Yoshihito and Professor Liu Qiusheng, for your insightful comments and suggestions. I am grateful for the time you have devoted to reviewing my dissertation. Your expertise and feedback have been valuable assets to my research and significantly improved its quality.

Thank you to Ms. Nakao Kanako for your unwavering support, care, and encouragement throughout my three years of study. I am grateful for your kindness and generosity to me and other international students. Your dedication to our success and well-being is truly remarkable. And I would also like to thank the rest of the teachers and staff at Kobe University for your support and assistance during my time as a student.

Thank you to my Japanese teachers, Associate Professor Saito Miho, Ms. Goya Akemi, Ms. Asada Keiko, Associate Professor Park Sooyun, and Ms. Ohsaki Keiko from Kobe University; and Ms. Ochi Toshiko, Ms. Higashino Masako, Mr. Konishi Koji, Ms. Nakase Etsuko, Ms. Kuno Miyuki, Ms. Abe Hiromi, Ms. Ueda Taeko, Ms. Maruo Kuniko, and Mr. Ohtani Hirohide from Ashiya Municipal Shio-Ashiya Exchange Center; for providing me not only with the best guidance in my Japanese language studies but also in life. Your words of wisdom and actions have been a source of guidance for me and have helped me to navigate the challenges and opportunities of life. I am forever grateful for your unwavering mentorship.

Thank you to my dad, Liu Dexin, and my mom, Huo Yaping, for your indispensable love and dedication over the past thirty years. Despite the challenges posed by COVID-19, which prevented me from returning home even once, you have been a constant source of motivation and inspiration, even from afar. Your devoted care and concern has meant the world to me.

Thank you to all of my friends in Japan, in China, in Korea, in Indonesia, in America, in Spain, in Myanmar, in Australia, in Canada, and all around the world for your steadfast friendship, encouragement, and companionship throughout my academic journey. I regret not being able to list all of your names here individually due to space limitations and not wanting to inadvertently leave any of you out. I am incredibly grateful for the opportunity to have had you as such wonderful friends and for the memories we have shared together.

I would also like to express my sincere gratitude to the Ministry of Education, Culture, Sports, Science and Technology (MEXT) for their generous support throughout my research. I am deeply grateful for the funding provided by the Japanese government scholarship, which has greatly aided my academic pursuits and allowed me to fully focus on my research.

Last, but certainly not least, thank you to all the kind strangers I have met along the way. Though I may not know your names, though our interactions may have been brief, and though our paths may never cross again, your thoughtfulness and compassion have touched my heart and left a lasting impression. I am grateful for the fleeting but beautiful moments we have shared.

Thank you, everyone.

For everything.