# Modular Human-Operated Radar Framework

Partial Dissertation


Presented by:
Wilhelm L. Carstens

Prepared for:
Dr. Simon Winberg
Dept. of Electrical Engineering
University of Cape Town

Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Masters of Engineering degree in Radar and Electronic Defence.

**August 2023**

**Key words:**
IoT, Signal Chain, MQTT, SaaS, Radar system framework, browser-based HMI, Leaflet, Progressive Web App

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This thesis/dissertation has been submitted to the Turnitin module (or equivalent similarity and originality checking software) and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

..... Signed by candidate .....

Wilhelm L. Carstens

# Abstract

This study investigates the development of operator-facing radar systems with contemporary internet technologies, such as the Internet of Things (IoT) and cloud services. The viability of modular designs allowing a high degree of adaptability is emphasised, given the inherent capabilities of IoT application-level protocols. The use of other internet technologies and services focus on the increased functionality, commonality, and flexibility they provide to modern integrated radar systems.

The investigation starts with an overview of operator-facing radar systems, detailing their current and near-future application, broad design considerations, common architectures and web resources available for their development. In evaluating various IoT protocols from literature, the MQTT protocol is selected and then experimentally analysed against pure transport protocols on consumer hardware, characterising their usage. Then, using these technologies, a common framework is designed and developed, alongside a browser-based Human-Machine Interface (HMI) that allows for general usability and performance testing.

These tests reveal the implementation to be adequate for many high-level uses, but at some expense to overall data latency and load, necessitating specific consideration where used. Furthermore, IoT protocols allow for distributed radar systems and highly adaptive single-flow signal chains without employing conventional server infrastructure.

Although the conceptual framework is not well suited for all radar uses, it does offer a versatile solution for various high-level applications, with future developments in IoT protocols showing particular promise.

# Acknowledgment

I would like to express my sincere gratitude to Dr. Simon Winberg of the Department of Electrical Engineering at UCT for his patience and encouragement to further my aims with this dissertation. I'm also extremely thankful to John-Phillip Taylor, of the same department, for repeatedly lending his ears and deep LaTeX know-how when things invariably got overly tangled.

# Contents

# List of Figures and Listings

# Nomenclature

## Acronyms

ADC ..... Analog-to-Digital Converters

ADS-B ..... Automatic Dependent Surveillance-Broadcast

AIS ..... Automatic Information System

AMQP ..... Advanced Message Queuing Protocol

API ..... Application Programming Interface

ATC ..... Air Traffic Control

AWS ..... Amazon Web Services

CoAP ..... Constrained Application Protocol

CPU ..... Central Processing Unit

CSS ..... Cascading Style Sheets

CSV ..... Comma-Separated Value

DaaS ..... Data as a Service

DDS ..... Data Distribution Service

DOM ..... Document Object Model

ECMAScript . European Computer Manufacturers Association Script

FAA ..... Federal Aviation Authority

FPGA ..... Field Programmable Gate Array

GIS . . . . . Geographic Information System
GUID . . . . . Globally Unique Identifier

HCI . . . . . Human-Computer Interaction
HFE . . . . . Human-Factors Engineering
HMI . . . . . Human-Machine Interface
HPC . . . . . High Performance Computing
HTML . . . . . HyperText Markup Language
HTTP . . . . . HyperText Transfer Protocol
HTTPS . . . . . Secure HyperText Transfer Protocol

IaaS . . . . . Infrastructure as a Service
IETF . . . . . Internet Engineering Task Force
I/O . . . . . Input/Output
IoT . . . . . Internet of Things

JSON . . . . . JavaScript Object Notation

LWT . . . . . Last Will and Testament

ML . . . . . Machine Learning
MOM . . . . . Message-Oriented Middleware
MQTT . . . . . Message Queueing Telemetry Transport
MQTTS . . . . . Secure Message Queueing Telemetry Transport
MQTT-SN . . . Message Queueing Telemetry Transport for Sensor Nodes

NEXRAD . . . Next-Generation Radar
NRT . . . . . Near Real-Time

PaaS . . . . . Platform as a Service
PCI . . . . . Peripheral Component Interconnect
PWA . . . . . Progressive Web App

QoS . . . . . Quality of Service

REST ..... Representational State Transfer

SaaS ..... Software as a Service

SDK ..... Software Development Kit

SPA ..... Single Page Application

SVG ..... Scalar Vector Graphics

TCP ..... Transmission Control Protocol

TCP/IP ..... Transmission Control Protocol / Internet Protocol

UDP ..... User Datagram Protocol

UI ..... User Interface

URL ..... Uniform Resource Locator

USRP2 ..... Universal Software Radio Peripheral 2

UX ..... User Experience

VM ..... Virtual Machine

WebGL ..... Web Graphics Library

WMS ..... Web Map Service

WSS ..... Secure WebSockets

XML ..... eXtensible Markup Language

YAML ..... YAML Ain't Markup Language

# Terminology

| | |
|---|---|
| **Broker** | IoT message-oriented middleware that serves as a host for the primary data model and to facilitate messaging. |
| **Client** | Computing node for the application used to access and control the system. |
| **Developer** | A subsystem developer serving as the framework's principal user. |
| **Data model** | An abstract semantic hierarchical model that depicts the specific relationships between defined data elements in a specific context. |
| **Framework** | Inherently modifiable foundation of shared libraries and resources providing generic functionality for targetted software development. |
| **Internet-based** | Specifically premised on the networking technologies that enable the internet, a global network of networks using standardized infrastructure and protocols for interconnection. |
| **Operator** | The client-side application user, often expected to exclusively interact with a wider system through a defined graphical interface. |
| **Platform** | General computing environment as a combination of the device type and operating system. |
| **Protocol** | A set of standardised syntax, rules and directives for data packing and unpacking, enabling effective data transmission between entities operating at the same computational layer. |
| **Server** | Program serving resources to client applications, e.g. web server providing the site resources to connecting browsers. |
| **Signal chain** | A data flow pattern defined by a chain of sequentially connected processing components that allows for a more sophisticated interpretation of incoming signals/data - becoming analogous to a classical architectural software pattern at a very high level (e.g. the Pipe & Filter or Hexagonal architectures). |
| **Web-based** | Specifically premised on the higher-level technologies in the information space operating over the internet that enables the World Wide Web. |

# Chapter 1

# Introduction

The functionalities required for radar systems to be fit for purpose have significantly expanded since their inception and continue apace with their expanding use. Radar system development has simultaneously seen a concerted shift to higher-level solutions, with numerous radar subsystems now implemented in software where general-purpose hardware performance increases have met and, sometimes even, exceeded what they necessitate. [1]

The various technological improvements have also resulted in earlier high-performance data processing, reducing the bandwidth needs of higher-level subsystems. This allows for successive stages to exhibit a high degree of interconnectivity, facilitating the use of modern software approaches that favour versatility.

Select contemporary internet technologies, especially Internet of Things (IoT) messaging infrastructures, have aspects that show considerable promise for high-level radar system use. Broader cloud-service offerings also provide numerous features that may expand the capabilities of many operator-facing radar system applications. [2, 3]

A wholly data-driven radar system framework allowing multi-operator access should now be viable when paired with a custom-designed Human-Machine Interface (HMI). A composite system built on this framework then enables the investigation of certain aspects prevalent upon simple radar systems, as applicable with select scenarios within a modern usage context.

## 1.1 Background

Radar systems allow for the detection of objects from reflected radio signals. The contemporary use of radar is varied and numerous, spanning from the dedicated detection and tracking of ballistic missiles in the military domain to the monitoring of movement to activate automatic door opening. Many of the more sophisticated uses of radar see defined roles performed by human operators, albeit ever less directly as the scope of machine assistance expands. [4, ch. 1]

Radar systems classically consist of transmitting and receiving components, with meaningful interpretation requiring various processing-intensive operations on the received signal as dictated by the system's intended purpose. These take the form of various specialised subsystems, each requiring the application of focused approaches and specific expertise, resulting in extensive modularization. Along with rapid advances in modern general-purpose computers, many subsystems are now implemented through software, considerably benefiting radar development from practises typical to this modern branch of technology. [2]

Less mundane radar systems typically have significant on-site performance requirements due to the high volumes of data and the nature of specialized algorithms they involve. Hence, processing activities often require dedicated computing devices with closed-off, low-latency, high-throughput communication channels providing direct access to radar system data and the use of a purpose-built environment. [2, 5]

Higher-level software also permits iterative development, and the availability of various open-source software-based modules allows for rapid prototyping of related subsystems [6]. Internet-oriented designs, in particular, offer benefits by allowing for the common dissemination of sensor data, along with the processes that enable their interpretation. [7]

IoT-based technologies, particular to this environment, have not generally been considered a viable foundation for radar system development - likely due to the inherent need to maintain a good quality of service through lossy communication channels and the various overheads this entails [2, 8]. However, IoT application-level protocols would organically allow for the complete reshaping of dynamic signal chains as needed [9] and present a further step in leveraging modern software technologies within the radar domain.

## 1.2 Objectives

This study considers the application of modern internet technologies to high-level radar system development. It will examine realistic radar system use cases and explore how they can be solved using commonly available tools and hardware.

The primary objective will thus be to determine how well IoT technologies, which offer dynamic data filtering capabilities, may serve as a foundation to current and near-future radar systems. These systems may also benefit from various web-based services and supplementary data technologies and so see their usage and comparative performance aspects explored as part of this study.

Therefore, this work will detail the design and implementation of a simple modular, distributed, dynamic, self-describing, scalable radar system framework using modern internet-based technologies. A client HMI will complement the framework by providing operators with access to system-specific controls and data.

The combined solution allows determining how effectively the conventional web domain's benefits translate to current radar system development and use. Furthermore, co-related technologies that increase the versatility of modern software-based radar system solutions, especially where ubiquitous in contemporary online environments, will be investigated.

Transfer characteristics are particularly consequential since they have a significant impact on the types of systems that are practicable. These characteristics include throughput and latency with specific payload sizes and the nature of bottlenecks encountered.

Due to the general diversity of modern radar system applications, a viable solution should accommodate a wide range of layout options and depth. These token layouts will allow consideration of basic implementations and signal chains consisting of numerous processing stages, allowing for an appropriate usability and performance measurement of operator-facing radar systems.

## 1.3 Problem description

Radar subsystems are increasingly implemented through distinct software-based modules, using the improved performance allowed by current general-purpose platforms. Increased modularization also allows for stratification of subsystems by their data-transfer characteristics. Many radar subsystems are now in defined task-driven chains, resulting in much-reduced data-transfer rates at higher levels.

Even so, subsystems are often in specialized architectures that prioritize aspects of performance above broad reuse and development versatility. Although modern internet technologies should now offer a viable basis for high radar system development in general and operator-facing radar systems in particular, the extent to which this applies is unclear.

Herein a particular need exists to determine the extent to which modern internet technologies (especially IoT-based) offer benefits to modern radar systems, which radar applications may now be conceivable using these technologies, and to what extent does their inherent flexibility apply, based on their performance and interoperability.

## 1.4 Scope and limitations

A comprehensive implementation utilising unprocessed radar data is not considered essential, given the primary goal of demonstrating the feasibility of a rudimentary but realistically defined system. Where an alternative option provides a reasonable means to show relevant capabilities to a sufficient degree, such may be employed.

Even though select web services will form part of this study, these services may not always be essential to a particular radar application. As such, the study will not be premised on an active internet connection necessarily being available, but will consider select aspects where considered relevant.

Consideration for deployment on High-Performance Computing (HPC) and the specific use of Machine Learning (ML) will be contextualized for potential future exploration but not investigated in detail here.

Low-performance hardware for subsystem implementations will not be a dedicated focus here, with greater emphasis on the use of current-generation hardware as would be commonly available for low-to-mid range radar systems. This assessment includes cloud-based solutions, as these are a crucial consideration with some implementations, notably distributed systems or where the time-shared use of costly resources is required.

## 1.5 Document outline

**Chapter 2** reviews the current literature to define a list of current radar system uses that require direct Human-Computer Interaction (HCI). It will also provide a discussion of traditional radar system components and architectures. Next, the available distributed networking technologies and web-based services that may be used in a high-level radar system will be studied for potential use. Finally, contemporary HMI design aspects will be explored, with due consideration to conventional HCI research.

**Chapter 3** outlines the specific steps that will be taken to determine the degree to which contemporary internet-based technologies may be used to implement a rudimentary high-level radar system. The chapter will broadly describe the approach to implementing a prototype system as required for the uses and environments previously described in the literature review, with subsequent steps to determine the functional and performance aspects using the actualized solution.

**Chapter 4** gives an empirical analysis of the characteristics of the protocols narrowed for usage based on the prior review of the literature. The results will be used to determine the specific throughput and latencies that may be expected from the protocols under consideration concerning their use in operator-facing radar systems, providing a starting point for the technical design.

**Chapter 5** details the design of a dynamic radar system framework that allows for a modular approach, in light of the general architectures defined in Chapter 2. A visual design for the accompanying HMI, tasked to ensure the primary functional use of a system built on the framework, will also be provided here.

**Chapter 6** considers the appropriateness of the resulting implementation for some general usage scenarios that apply to modern radar systems. It will also analyze some performance aspects pertinent to radar elements, further detailing the approach's viability.

# Chapter 2

# Literature Review

This chapter details some of the broader aspects that allow for the foundation of a general radar system framework using internet technologies, with a holistic approach that considers both current and future usage expectations.

Herein various operator-facing radar system applications in the military, commercial and engineering domains are explored for overarching features and general usage trends. Modern radar system design and architectures are also considered in light of classical systems, emphasising the broader advances in general computing, design trends, changing expectations, and benefits gained as a result.

The subsequent sections examine a selection of modern internet technologies for their potential use in radar systems as outlined. An initial focus on viable IoT application-level protocols provide insight into their characteristics, informing a reduced selection of protocols for use in the eventual framework based on comparable aspects. Various web services and related technologies that provide an expanded feature set and implementation versatility is considered, specifically for the previously specified radar applications.

Further considerations and technologies outlined in these sections drive an assessment of the common technology stacks for the integrated HMI. Generally accepted fundamentals for direct interfacing with an operator are examined, taking into account fundamental HCI features and some of the contemporary design trends in the consumer space that follows therefrom.

## 2.1 Context of operator controlled radar

This section describes current operator-controlled radar system usage scenarios and trends. Due to the nature of general technological advancement, radar has lost its dominance as a solution in domains where it was previously considered a monopolized solution (e.g. the rise of cooperative solutions in commercial air traffic management), becoming a component of a wider selection of systems. For any meaningful consideration of radar systems and their technical implementation, a high-level assessment of the pragmatic application of radar in present and near-future systems, as provided in contemporary literature, is necessary.

Both military and commercial environments are considered, as they often show conformant requirements due to broadly similar performance and safety concerns. With these systems utilized by specialist operators for broader situational awareness and assessment, they often have strict Near Real-Time (NRT) requirements. [10, 11]

The increased automation of operator tasks has resulted in a sustained trend toward interpretive data presentation, away from direct evaluation of radar data. Nonetheless, operators in specialised roles face growing workloads due to the increased number of associated systems with which they must interact to perform their duties efficiently.

As a result, operator-controlled radar systems are increasingly moving toward truly integrated solutions that intelligently combine various primary and secondary radar systems, as well as supplemental data sources. To that end, there is an increased demand for these systems to consider the operator's overall workload with ever-higher abstraction to ensure that information is conveyed efficiently. [12, 13]

Given that specialist operators frequently serve as key decision-makers, various radar applications with ongoing HCI needs can now be identified as providing guidance for prospective radar usage scenarios or in describing the broad domain.

### 2.1.1 Air traffic management

Air Traffic Control (ATC) systems are a current driver of radar as part of an integrated system within both the military and commercial domains, mostly with the advent of cooperative solutions.

The locality of conventional radar systems often leads to large surveillance gaps, hindering the continuous monitoring of commercial flights. This has led to increased integration between different ground-based ATC systems and other satellite-based technologies, such as Automatic Dependent Surveillance-Broadcast (ADS-B) systems.

With commercial ATC operations seeing progress towards this integrated whole, the use of radar as a primary surveillance consideration is in decline. Radar is likely to remain in the mix of solutions, but mainly as a fallback and to provide additional capabilities, esp. with the increase of potentially uncooperative entities, such as drones, in the modern airspace. Ground-based radar is more likely to see increased use in non-traditional applications, such as monitoring aircraft and airport vehicles' surface movement and detecting wake vortices to determine safe take-off separation distances. [4, p. 51]. [14, 15]

The local air picture provided by weather monitoring systems is essential to the effective routing of aircraft. Aircrews may even remotely access these systems for pre-flight planning or route optimization, as with Traffic Aware Strategic Aircrew Requests (TASAR) [16] encompassing ground-based sensors into the integrated whole of available sensors. [17]

### 2.1.2 Airborne platforms

Aircrew in the military and commercial domain often find themselves in highly specialised multi-system roles. These require full situational awareness in various scenarios, with some reliance on platform mounted radar systems. A task-focused approach is essential, as with precipitation monitoring and its combined display for continuous flight planning. [18]

With the potential presence of non-cooperative entities in complex environments, military aircraft use radar systems for diverse purposes. Monitoring and target tracking systems are typical, along with advanced collision and ground avoidance systems (e.g. Terrain Following Radar (TFR) and Ground Proximity Warning Systems (GPWS)) that enable coordinated missions at low altitudes.

Some sensor intelligence operations may see aircraft modified or even specifically manufactured to perform distinct surveillance functions. [19, 12] Airborne Early Warning (AEW) platforms exhibit similar challenges to traditional ground-based systems, owing to larger aircrews and higher operator specialisation than with multi-role aircraft. [20]

### 2.1.3 Military application of surface-level systems

Military air and coastal surveillance systems provide the foundation for early detection and missile interception systems. These systems are sometimes entrusted with safeguarding assets of vital economic and strategic import, but typically also form an essential part of the defensive capabilities of stand-off platforms such as naval vessels.

Radar systems in this domain often form part of an even greater system of interconnected systems, providing a comprehensive picture at higher echelons. Such is required not only for sensor redundancy in an environment rife with potentially hostile entities but also to allow for the implementation of an effective layered defence.

At far ranges search radars provide both early warning and preliminary target characterization. These are crucial to long-range missile defence, with targets showing specific signatures even at great distances. Such systems also provide insight into the nature of an opponent's tactics, which is vital to the timely preparation of an appropriate response. [21], [4, p. 41-42]

At mid-ranges higher resolution surveillance systems are required to ensure target distinction and specific classification abilities within their area of responsibility. This provides decision-makers with increasingly detailed target information as an ongoing situation unfolds, further aiding decision making.

At close ranges, the response is limited to local interception and weapon-based tracking systems. [4, p. 46] Even in this context, these systems feed into larger decision support structures to ensure optimal operator performance. Artillery detection systems, for instance, may monitor both inbound and outbound projectiles to aid engagement awareness and determine threat sources on a wider front. [4, p. 48]

Ground-Based Air Defence (GBAD) systems exemplify such an interconnected system of systems, requiring a wide variety of radar systems for redundancy and task specialisation. Again, there is a tendency toward allowing systems and modules to cooperate to give a more comprehensive picture of the airspace with fused sensor data [22, p. 69].

### 2.1.4 Monitoring of non-classical targets

With rapid advances in computing, various radar subsystems have seen significant increases in capability. Tracking systems have especially benefited, with high-level solutions allowing NRT recognition and tracking of even small and slow-moving entities. [22, p. 98]

Radar systems that work in tandem with perimeter defence systems, such as those used for intruder detection and national border protection, may be adapted for animal conservation. Tracks here behave irregularly compared to traditional radar targets, so new predictive modelling and tracking advances also greatly benefit these systems. [22, p. 42-43]

The typical obligations of military radar systems have seen them being favoured as a tool for certain environmental research – as where high-altitude targets must be detected or vast distances observed in low-visibility conditions. Monitoring large-scale migrations of flying species is one such use, offering significant insights into swarm behaviour (which may be fed into air traffic systems). [23] Such supplementary usage benefits from versatile systems that can be adapted to some degree as need dictates.

### 2.1.5 Engineering and development

The impact of modularization and ease of integration to further system development is an often overlooked aspect of modern radar systems. The military domain, in particular, sees the need for constant innovation in radar technologies due to the common desire to improve the capability to detect potentially hostile forces whilst simultaneously reducing the probability of own force detection [24].

To some extent, modern radar systems now resemble generic technology platforms rather than static products. The versatility afforded herewith is especially beneficial due to the rise in system complexity necessitating quick assessment of innovative procedures on prototype and technology demonstration systems. This rapidly evolving nature often results in instrumentation lagging behind what is required to validate system operation, such that high-level tooling is now often a part of their development. [24]

The improved monitoring and analysis afforded by radar systems also see their use in a supporting capacity for various engineering and technology development tasks of other

systems, to which they must keep tread. Such is especially the case at missile test ranges, where these systems provide essential trajectory data, in addition to the crucial role they play in range safety [4, p. 46], [25]

## 2.2 General radar systems

This section considers the broad requirements of radar systems in light of current technological developments and general design trends. To this end, it briefly reviews aspects central to all eras of radar systems development, which in turn contextualizes the broad characteristics and architecture of current and near-future implementations. Hereby it aims to ensure that any proposed framework is fit to the domain of radar systems development.

Radar is classically defined as a technique for detecting and ranging objects from reflected radio signals. It requires the timely detection of a drastically weakened reflected signal in angle and range within a complex extended environment. To this end, sensitive receiving elements are used alongside high power transmitters and subsystems that allow for the timely interpretation of high volumes of data. [4, p. 1-4], [26, p. 1-3]

Figure 2.1 presents a classic radar system block diagram, as presented in Merrill I. Skolnik's influential "Radar Handbook", which was first published in 1970. Here Skolnik notes the prevalence of the antenna and display components in the media's portrayal of radar systems [27, p. 4.1] – an incomplete picture that seemingly persists to this day.

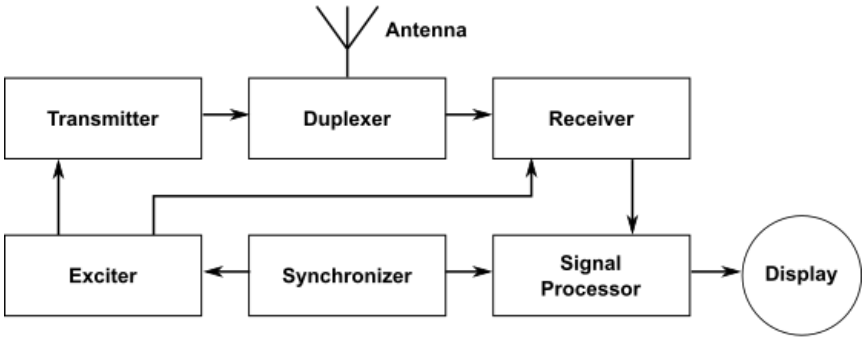


Fig. 2.1. Classic radar system block diagram, adapted from [27, Fig. 4.1]

The emphasis leans heavily on the lower-end analogue components of the pulsed radar system in figure 2.1, with a high degree of intertwining between its principle components.

An exciter is responsible for waveform generation for the transmitter and receiver components, with a duplexer to switch between their usage of the antenna. A centralised synchronizer's clock provides overall timing, even to the signal processor tasked with further interpretation of the returned waveform. Here, these modules do not exhibit a clear compartmentalization in accordance with the disciplines involved in their development.

Traditional radar systems are the culmination of many specialised fields, including radio frequency, mechanical, electrical, and electronic engineering. These are all still relevant in the development of modern radar systems, but along with rising complexity, particularly extensive specialization has occurred in the field of electronic engineering. [1, 2]

These specialized disciplines have mostly focused on capabilities that stand to gain the most from advances in computing capability, such as signal processing and subsequent interpretation to form tracks. Further specialisation seems probable with the advent of ML technologies, with interpretive modules even capable of behavioural target analysis [28].

### 2.2.1 Software implementations

Radar system development is not isolated from the wider engineering environment and technological trends, but rather benefits greatly from them. Purely software-based subsystems are increasingly prevalent due to an increased availability of generic computing modules, which exhibit ever greater processing capabilities.

There is a clear motivation for increased modularization of radar system components given the projected current and future uses of operator-facing radar systems (see section 2.1). It has resulted in self-contained components that can provide specialised capabilities to various connected systems. Given their black-boxed nature and high degrees of abstraction, such components may even see use outside of their initial design considerations.

The shift to software-based implementations can also be seen in modern radar system architectures, where subsystems frequently appear as successive processing stages within a sequential data flow. It is largely premised on separating analogue reality from the purely digital domain in which software-based solutions operate.
Such is the case with software-defined radar, where an analogue-to-digital conversion stage allows for the majority of real-time sensor data interpretation to be performed on digitised

data streams. [2] This is achieved by implementing the various hardware components (e.g mixers, filters, modulators, demodulators, detectors, etc.) in software on a computer or other programmable device [1].

Figure 2.2 represents a simplified modern radar system block diagram. It is comprised of highly abstracted components with its rigidly defined modules and a sequential architecture of particular note for the purposes of this study.

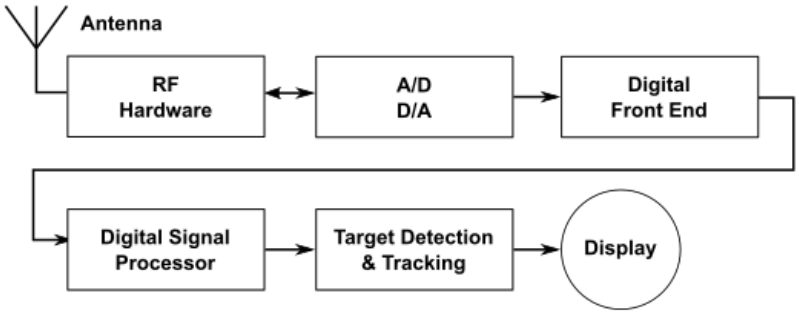


Fig. 2.2. Modern radar system block diagram, adapted from [6, Fig. 1] and [29, Fig. 9]

The inherent reusability of software-based solutions, combined with the availability of open-sourced modules, allows for an immediate focus on higher-level functionality. Higher levels of abstraction provided by software implementations also result in shorter development times at a lower cost. Additionally, the strong modularity typical of software architectures allows for highly controlled subsystem testing and debugging.

Software implementations are highly adaptable, even enabling dynamic reconfiguration of production systems in reaction to changes in the geophysical environment or when specific unknowns become problematic. Internal metrics and knowledge-based processing on input from other sources can thus be used to refine radar behaviour or even dynamically affect fundamental characteristics of the radar system as a whole. [2, 6]

High-level software-based implementations mitigate the overall costs and risks associated with custom-made or specialized hardware. Due to their generic nature, a single computational resource can be reused or shared by multiple systems and subsystem components. This enables the low-cost implementation of multi-purpose radar systems as well as the combination of multiple systems on space-constrained platforms. [1, 2]

Using general-purpose hardware increases processing overheads, lowering the effective radar

resolution, resulting in faults that require mitigation logic. The widespread availability of general-purpose computers allow for upgrades as needed or even relying on future generational improvements. Further measures include local parallelization, compute shaders, or task spanning across multiple distributed computing elements.
Special care should be taken with distributed systems, as these suffer from network latency issues and other throughput limitations. This can be mitigated by having subsystems co-located on a platform, but it does limit the overall reconfigurability of such a system. [2]

The large memory stores on general-purpose computers also provide various benefits. This includes enabling radar systems to build a picture of an environment over multiple scans, providing digital clutter subtraction for improved target detection [1]. A digitized copy of the transmitted waveform can also be stored and used in subsequent signal processing, negating the need for implicit knowledge of the waveform. Insufficient memory typically leads to a lower resolution and (depending on implementation) mirrors processing considerations.
Digitised data can also be easily archived using commercially accessible storage solutions. As such, a sufficient amount of secondary storage allows for radar data to be recorded for later playback, debugging or extensive analysis through specialized computing devices. [2]

### 2.2.2 Software architectural patterns

System components can generally be classified by the broad data flow patterns they exhibit in fulfilling specific operational needs. These patterns can then be combined as parts of a wider pattern. The Signal Chain architecture is one such overarching pattern. Within radar systems it sees a specific implementation with individual components classified as Generator, Selector, Transforming Element and Consumer blocks. [2] It is the primary signal processing software design for modern software-defined radar systems, combining aspects of the Hexagonal architecture [30] into the Pipe & Filter architecture [31].

A single signal chain is implemented by having one or more Generator blocks feed into Selector and Transforming Element blocks over a common set of channels. These blocks serve as successive processing stages that allow for data filtering and manipulation, eventually terminating in a Consumer block. Figure 2.3 shows a simple signal chain as presented by T.Grydeland *et al.* in volume 23 of the "Annales Geophysicae", with minor

adjustments to demonstrate Selector block use. Sections 2.2.2.1- 2.2.2.4 go through each of the blocks outlined here in further depth.

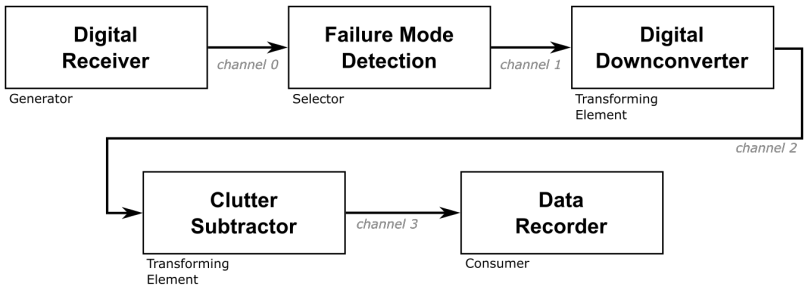


Fig. 2.3. Representative signal chain using default blocks, adapted from [2, Fig. 2(b)]

These blocks can be combined to create complex signal processing systems, allowing for the implementation of various solutions. The modular nature of such an implementation necessitates that data provided for processing be limited to pre-defined acceptable types at input and output. The sequential processing of data results in overall system performance being limited to the slowest processing stage, although parallel computing may be selectively used where delays are a cause for concern. [2]

#### 2.2.2.1 Generator block

Generator blocks may be tasked with obtaining and formatting externally sourced data for use in a signal chain, given the strict requirement for known data types with this architecture. Alternatively, a Generator block may directly create simulated data for testing; allowing for the practical assessment of functional capabilities or a comparison of different mechanisms with comparable data sets. [2, p. 114]

#### 2.2.2.2 Selector block

Selector blocks allow for the filtering of data. It not only reduces network usage requirements by limiting transferred data to a minimum, potentially even throttling data completely, but also detects and drops spurious and/or corrupted data. This selective data filtering eliminates the need for subsequent blocks to inspect incoming data, instead directing processes to their intended purpose. [2, p. 114]

#### 2.2.2.3 Transforming Element block

Transforming Element blocks manipulate specifically typed input data, which is then passed further along the signal chain. It is typically involved with computationally intensive tasks, performing traditional signals processing work such as digital filtering, numerical mixing, Fourier transformations and signal correlation. Transforming Element blocks may be merged or split depending on implementation flexibility or performance considerations, e.g. local parallelism of tasks where bottlenecks become apparent. [2, p. 114-118]

#### 2.2.2.4 Consumer block

With no subsequent processing, Consumer blocks serve as termination points in a signal chain. HMIs function as Consumer blocks by providing data to an operator in the form of direct visualizations such as tracks, plots or even internal system metrics (e.g. representations of data transfer and processing load). Consumers may also take the form of data recording elements, allowing for the permanent storage of data on disk or networked storage services. [2, p. 118-119]

### 2.2.3 Radar subsystem development

Modern radar systems are typically implemented as a combination of specialized computing elements, such as Field Programmable Gate Arrays (FPGAs), and general-purpose computers. Implementing a greater amount of the system's logic at a higher level allows for rapid development cycles, improved flexibility and significantly reduces costs.

The software patterns applicable to radar system development are not fundamentally tied to any particular programming language or abstraction level. [2] Software implementations usually take the form of an organised collection of software programs, with each program potentially written in a different programming language.

Significant parts of modern radar systems can now be realized through high-level languages such as Matlab [32], indicating a distinct trend towards higher levels of abstraction in modern software-defined radar systems. Using a higher-level programming language can offer specific implementation benefits, as it allows for prototyping of complex algorithms

and experimental functionalities at a reduced cost and effort.
Where processing bottlenecks or power constraints are then identified, these can be targeted for implementation at lower levels of abstraction or even FPGAs. This process allows for a selective investment of effort, often without duplicating labour – as techniques and algorithms in higher-level prototypes still apply when actualized at a lower level.

The software engineering field's widespread influence on radar system design is likely to continue, as evidenced more recently with the use of neural networking for radar-specific problems [28]. This is expected to result in the addition of even more functional elements to future radar systems, as well as an increased modularization within current subsystems.

Modern software designs' versatility is due, in large part, to flexible data interchange formats. These allow for simplified interaction with external systems, as specific data can be extracted from a larger structure as needed. Subsequent additions to these structures can also be made without interfering with active system operation.

Flexible data interchange formats of note include the schema-based Comma-Separated Values (CSV) [28], as well as the self-describing eXtensible Markup Language (XML) and JavaScript Object Notation (JSON) formats [33]. With generic interfacing between Signal Chain blocks preferred [2], such formats are encouraged in absence of other considerations.

## 2.3 Inter-module messaging on radar systems

This section narrows down the technologies that may be used to facilitate messaging between component blocks. The particular qualities of allowed message carriers will significantly impact the eventual framework, necessitating careful review. Optimal messaging mechanisms should be viable throughout the entire Signal Chain and support the transfer of both control and data messages. These message types can be regarded as distinct, with data messages favouring a streaming approach more so than control messages, which are intermittent but of high priority.

Multicast (one-to-many) messaging permits multiple entities to listen in on a common data carrier, which is critical to the Signal Chain architecture. This is due to a data stream

potentially having various loosely coupled uses (e.g. concurrent display, recording, and processing). It enables highly scalable signal chains and circumventing specific bottlenecks, while also enabling live testing and analysis without risking core functionality.

Repackaging of data for use on the common carrier causes processing delays, which becomes problematic with long signal chains; distributed nodes may also lead to flow control issues. As such, message flows must be carefully considered to avoid data starvation due to saturation or delays in other elements. Limiting rates to the slowest part in a flow ensures stability, but requires reliable messaging with adequate throughput. [2]

To be effective outside of tightly controlled environments, the solution must be resistant to common communication issues such as high throughput fluctuation, connection interruptions, and message fragmentation. Even so, the solution should offer sufficient throughput and latency for high-level data transfers (e.g. plots and tracks), with a separate approach only considered where high throughput is demanded.

Given that several independent entities may need to interact with system messages, the core communications technology should be mature and pervasive across a wide range of programming languages. It should also provide a well-defined interface while not being overly constrained in terms of the messaging patterns that can be used or emulated.

To enable common implementations, chosen protocols should ideally be natively supported, or otherwise readily accessible, in a broad range of computer languages, with ECMAScript, Python, Matlab, C, and C++ being particularly noteworthy.

The rich network of interconnected systems that underlie modern internet infrastructure has driven fundamental shifts in communications technology. These shifts have in turn led to a new generation of application-level protocols being adopted to optimize messaging between networks of highly distributed systems [34].

The aspects of messaging within modern radar systems, as detailed above, are now considered in light of the wider throughput requirements, potential fallback mechanisms and viable IoT protocols.

### 2.3.1 Classic absolute throughput constraints

To contextualize the various message throughput rates presented in this section, some of the constraints on messaging within typical software-defined radar systems is considered.

Digitized radar data is provided by Analog-to-Digital Converters (ADCs) as contiguous sample blocks; along with some overhead for framing and meta-data, such as timestamps. Assuming an individual sample size of 2 bytes, a representative ADC (e.g. GNU Radio's Universal Software Radio Peripheral 2 (USRP2)) manages a rate of 100 MSamples/s, or 200 MB/s, to the system.

In considering the use of general-purpose hardware solutions, external interfaces can be seen to present significant throughput constraints. Gigabit ethernet is only able to transfer samples at a potential upper limit of 62.5 MSamples/s – well short of even the low throughput USRP2 ADC. In comparison, internal interfaces are less of a concern, with Peripheral Component Interconnect (PCI) Express able to carry rates of up to 125 MSamples/s. PCI Express also scales linearly with the number of available lanes, with PCI Express x8 able to handle a message rate of up to 1 GSamples/s. [1, 2]

### 2.3.2 Standard internet protocols

The internet is premised on the transfer of datagrams through a number of interconnected of networks. [35] Such is accomplished by the encapsulation of data by a layering of inter-process, inter-application and inter-network protocols – allowing for senders and receivers at each level to focus on aspects of messaging at their operational level [36].

The major transport layer protocols to consider are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP) (see Appendix A.1 for more details). These protocols serve as the basis for IoT protocols, but offer benefits when used separately.

TCP offers flow-control measures, this lowers dropped packet counts in comparison to UDP, but result in overheads that reduce throughput. TCP messages may also be fragmented to facilitate transmission over underlying connection layers, further reducing efficiencies [37, 35]. Generally measures to improve message reliability reduce network performance, so acceptable data loss rates must be carefully considered to requisite transfer rates. [38]

### 2.3.3 IoT application layer protocols

This section details common aspects of the application layer protocols upon which IoT technologies are premised and so form a core foundational aspect of the framework. A listing and review of protocol options is included, with the performance metrics and available messaging patterns of each of particular note to further considerations.

Modern Internet of Things (IoT) messaging protocols offer tantalizing prospects toward standardized distributed messaging. These are reliable multicast transport mechanisms but come with significant performance overheads, introducing further complexities to a radar system's design. As such, these intricate protocols are unsuitable for the transfer of unprocessed data. They are, however, well suited to the transfer of control messages and high level processed data (i.e. plots and tracks, not raw data from a Digital Front-End).

IoT networks are regarded as the next evolution of the internet in the consumer market. It emphasizes the gathering, analysis and distribution of data from a large number of, typically small, distributed sensors – resulting in scalable networks with a high degree of flexibility [39]. Systems based on this technology are sometimes data-intensive, with reliable, low-latency, high-performance messaging between multiple entities expected [40].

Message-Oriented Middleware (MOM), which is common to many IoT protocols, manages data exchange and provides inter-node message routing and filtering. The attributes that may be used to match messages range from basic keying by a particular topic to thorough traffic and content analysis across several messages. MOM can either be implemented through a hosted broker service or on a peer-to-peer basis between nodes.
In general, broker based protocols allow for more varied and sophisticated routing patterns than peer-based protocols. Routing may take the form of simple send-receive patterns, e.g. 'Request-Response', or as more involved multicast patterns, e.g. 'Publish-Subscribe'. [40]

The 'Request-Response' pattern is one of the most basic communication paradigms and is common in most client/server architectures. It allows for client nodes to request information from a centrally managed server node, which then processes the request and returns a response message.

The 'Publish-Subscribe' pattern allows for loosely coupled, distributed, many-to-many, asynchronous communication between nodes. Event-based architectures are often used

to implement this pattern. MOM may provide queues that allow for store-and-forward mechanisms that facilitate messaging between producers (publishers) and consumers (subscribers), even where they are not connected simultaneously. A subscriber can indicate interest in particular events/topics to the MOM, which then selectively directs messages to that subscriber – so publishers and subscribers do not need to be mutually aware. [38]

The following subsections provide an overview of four of the most prominent standardized, platform-independent IoT protocols. These all allow for unstructured data and do not have onerous restrictions on payload sizes. They also allow for configuration of the Quality of Service (QoS) levels [38, Table 1], providing a measure of control over the weighing of reliability to performance.

Specific performance measurements as provided by Cui [41], whose analyses focuses on single-core Raspberry Pi use, are detailed for comparison. Cui uses short message bursts (10 consecutive messages) and continuous messaging (1 000 consecutive messages) to examine protocol behaviour. [41] Messages under 100 B may be considered small and representative of high-level data such as plots or tracks. Messages of around 100 kB may be considered large and representative of packed data payloads or even small image transfers.

#### 2.3.3.1 Advanced Message Queuing Protocol (AMQP)

An OASIS group standard that was originally designed for application messaging to a central service. AMQP has an 8 B header length and exclusively uses TCP as transport. Depending on which protocol version is used, it can make use of either broker or peer-to-peer MOM. The availability of a broker on older versions allows for use of the basic 'Publish-Subscribe' pattern as previously described.
It is seen as a good pick where heterogeneous systems and environments may be in play, as it allows for various platforms/languages to be used in concert. [38]

AMQP is a rather heavy protocol and is not recommended for use on performance constrained devices [38]. Message encoding/decoding involves a relatively large amount of processing with short burst messaging (∼90% CPU usage) but processing tapers off with continuous messaging (∼45% CPU usage). It has moderate memory requirements that stay relatively stable over various usage scenarios (∼10 MB). [41, p. 46-47]

The protocol offers reasonable latencies for short burst messaging with small messages (∼105 ms) but tends to be rather slow with larger messages (∼780 ms). A continuous transfer of small messages is seen to average at around 38 ms. [41, p. 42-45]

#### 2.3.3.2 Data Distribution Service (DDS)

An Object Management Group (OMG) specification that is widely used in select industries. DDS has a 16 B header length and can use either TCP or UDP as transport. The protocol does not make use of a broker, but rather decentralized peer-to-peer communication. Even so, it still allows for the more involved 'Publish-Subscribe' pattern to be used. [38]

DDS is a somewhat heavy protocol and care should be taken with performance constrained devices. Message encoding/decoding involves a relatively large amount of processing with short burst messaging (∼16-36% CPU usage) but processing tapers off somewhat with continuous messaging (∼5-30% CPU usage). It has rather high memory requirements that can rise significantly with message size and number (∼16 MB, but seen to increase to 250 MB with larger payloads). [41, p. 46-47]

The protocol shows rather high latencies for short burst messaging (∼270 ms) but results in a relatively flat latency with larger messages (∼390 ms). A continuous transfer of small messages tends to average at a respectable 7 ms. [41, p. 42-45]

#### 2.3.3.3 Constrained Application Protocol (CoAP)

This protocol was co-developed by the Internet Engineering Task Force (IETF) and ARM as the standard for future IoT applications. CoAP has a 4 B header length and uses UDP as transport. The protocol does not make use of a broker, but rather decentralized peer-to-peer communication – with all peers required to use IPv6 addressing. It allows for the use of the 'Pull', 'Push' and 'Observe' (analogous to 'Publish-Subscribe') patterns.
CoAP is a relatively new messaging protocol that is premised on the Representational State Transfer (REST) architecture, which allows for interoperation with HTTP without the need for complex translator logic. [38, 42].

CoAP is a very lightweight protocol that is suitable for use on performance constrained

devices. Message encoding/decoding requires a very low amount of processing with short burst messaging (<1% CPU usage) it does however show a slight increase with continuous messaging (∼2-3% CPU usage). It has very low memory requirements that stays flat with an increase in message size and number (∼3 MB). [41, p. 46-47]

The protocol offers fairly low latencies for short burst messaging (∼25 ms), which rises noticeably with larger messages (∼250 ms). A continuous transfer of small messages tends to average at a very reasonable 19 ms. [41, p. 42-45]

#### 2.3.3.4 Message Queueing Telemetry Transport (MQTT)

This OASIS group standard was initially developed by IBM and then released to the open-source community. MQTT has a 2 B header length and exclusively uses TCP as transport; there is however a variant, MQTT for Sensor Nodes (MQTT-SN), that can be used over UDP. MQTT uses a broker as its MOM and makes exclusive use of the 'Publish-Subscribe' pattern. It is a popular [38] messaging protocol that is well established and has extensive third party tooling [41].

MQTT is a lightweight protocol that is considered suitable for use on performance constrained devices. Message encoding/decoding requires a fair amount of processing with short burst messaging (∼10% CPU usage) and shows a wide span with continuous messaging (∼3-20% CPU usage). It has very low memory requirements that stays flat with an increase in message size and number (∼1-3 MB). [41, p. 46-47]

The protocol offers somewhat reasonable latencies for short burst messaging (∼90 ms), which rises significantly with larger messages (∼850 ms). A continuous transfer of small messages tends to average at a latency of about 58 ms. [41, p. 42-45]

Performance metrics change significantly for MQTT-SN, which compares very favourably to CoAP – which can mostly be attributed to both using UDP as transport. MQTT-SN averages 30% faster than CoAP over time, but with less consistency than CoAP. [43] However, MQTT-SN support in widely accessible brokers is limited, and alternatives, such as MQTT-SN to MQTT gateways, are platform-specific. [38]

### 2.3.4 Protocol selection and considerations

Here the previously reviewed protocols are reduced to candidates considered as viable for further investigation and finer selection. A more detailed reasoning to the selection of these protocols is provided in subsequent sections.

UDP and TCP (see section 2.3.2) provide somewhat distinct usages and are both seemingly viable at this stage, necessitating further consideration before a final selection. Given the direct availability of a transport layer protocol to the framework, the use of higher-level IoT application layer protocols (see section 2.3.3) can be focused to where they can be most meaningfully applied. Here the MQTT protocol is selected for further investigation to the use of IoT-based messaging due to its common use.

#### 2.3.4.1 Lower-level messaging

IoT protocols are unsuitable for transferring large amounts of radar data, necessitating the use of an alternative protocol for high throughput. To maximise network use, payloads are often packed; with application to scanning radars, this results in data being sent in angular blocks, leading to fixed interval message bursts.

Where a higher message throughput is required, standard internet protocols can be used for more targeted communication. Whether TCP/IP or UDP should be preferred is often considered situationally dependent.

Limited losses may not necessarily be a critical issue, particularly if the occurrence of losses can be identified and statistical analysis applied to data received over multiple scans. Here the prioritization of performance and long term optimization may well see UDP favoured due to its leaner implementation and lower processing needs.

On the other hand, where lower throughput is considered an acceptable trade-off due to any data losses leading to system fragility, or network communication not being dependable, TCP would generally be preferred.

#### 2.3.4.2 Higher-level messaging

With signal chains effectively filtering and compressing data at each stage (see section 2.2) away from the ADC, high-level data requirements are significantly reduced.

For a pure web application, exclusive use of transport layer message protocols is insufficient. Effective messaging would usually be assumed with HTTP (HyperText Transfer Protocol) (see Appendix A.3.1), but this does not allow for the push-based messaging required here. To this end, WebSockets (see Appendix A.3.2) provides push-based messaging to connected clients.

Given their ease of use, broad applicability, general performance, and computational requirements, CoAP and MQTT appear to be suitable choices for high-level use. MQTT is selected due to familiarity with this protocol and the commonly used mosquitto broker.

MQTT is seemingly not a good fit at first glance, but this picture changes somewhat when considering the use of MQTT-SN. It offers a viable upgrade path should greater performance needs justify the accompanying increase in effort. However, it does result in the absence of TCP's flow-control mechanisms [43] and certain implementation constraints.

Other factors in MQTT's favour lie in its stability, maturity, and extensive industry support to CoAP [38]. CoAP does remain quite promising though, especially for the transfer of smaller payloads at higher rates – but is still evolving and is not seen to have reached MQTT's wide adoption yet [38, p. 23].

With Last Will and Testament (LWT) capability as a typical feature of common MQTT brokers [44, 45], fully-featured control schemas may be given to numerous clients without the explicit requirement for a separate, intermediate management system.

The presence of a broker that provides store-and-forward functionality in the form of persistent messages allows for more architectural flexibility, particularly when accessing the last active configuration or state data of offline components for debugging.

In considering latencies of at most 250 ms for high-frequency transfers, MQTT payloads should preferably be limited to about 16 kiB (16 384 B) [41, p. 42-45]. Where MQTT payloads exceed 20 kB, such as with small images, latencies on low performance hardware can be prohibitive and transfers should be limited to once every few seconds.

## 2.4 Internet-based services

This section considers the use of internet-based solutions typically associated with, but not necessarily constrained to, the internet. These environments and services are shown to offer various advantages to the implementation and use of integrated radar systems.

Multiple users may utilise the same application service or infrastructure concurrently using multi-tenant solutions. When coupled with other standardisation efforts, this centralised approach allows the upkeep and expansion of a core feature set [9]. Various specialised features can therefore be diffused to a large audience with these technologies.

As seen in section 2.1, modern radar systems are of limited use in isolation, with even active monostatic radar systems requiring supplementary data and the collation of various external data sources and functionalities [12]. The ability to interact with wider web-based solutions allows for feature-rich implementations that can significantly enhance a system's utility without the need for dedicated on-site infrastructure or specific expertise.

### 2.4.1 Distributed computing environments

Cloud computing enables ubiquitous access to virtualized environments and pooled resources at various levels of abstraction in a unified workspace. It allows the acquisition of significant computing capabilities without the cost, risk, or effort associated with the procurement and maintenance of on-site infrastructure. Usage may also be scaled up and down on-demand, providing a versatile solution that many online services depend upon.

Infrastructure as a Service (IaaS) is the base layer of cloud computing, providing access to virtual machines, storage, servers, and network infrastructures. It essentially enables the leasing of hardware from a limited selection of resources. IaaS only provides a virtualized infrastructure environment, with platform and application setup left to users.

Platform as a Service (PaaS) adds an abstraction layer over IaaS that presupposes the managed availability of various resources. These platforms may be generalized or offer select access to specific runtime environments. Users need to install and configure their applications in the provided environment, with limited access to the underlying layers. As such, application setup may involve the use of a proprietary API. [46, 47]

#### 2.4.1.1 Remote hosting

IaaS and PaaS provide rapid provisioning of a widely accessible, robust, and scalable environment that may be used for deployment of specific radar subsystems. While these hosted environments often allow for considerable customization, they may be very rudimentary, necessitating the deployment of the entire application stack.

The latency and overheads introduced by these services may lead to a bottleneck that excludes their use for certain system functionalities. It does, however, provide secure access to other cloud services, as well as data redundancy, uptime guarantees and robust hosting capabilities [46].

#### 2.4.1.2 High Performance Computing (HPC) clouds

HPC clouds allow for the use of specific, highly performant computing resources that may not be commonly available to users. Cloud-based HPC clusters may be dynamically modified to accommodate additional instance types, improved interconnecting network throughput, increased memory, and faster processors. [5, 48]

While its applicability is largely context-dependent, it is generally considered as well-suited for applications that require a large number of parallelizable post-processing operations on cloud-based data [5]. This would imply a limited use for real-time signal processing but a substantial advantage for later, higher-level processing stages – such as weather data and certain types of target analysis.

Given the potential need for these specialised computing environments with an expanded solution, prominent cloud ecosystems such as Amazon Web Services (AWS) and Microsoft Azure are considered for common usage due to their scalable HPC clusters. Furthermore, problem-solving environments may be created by giving tools, such as the Matlab Compiler Runtime Environment (MCRE), access to these cloud-based HPC clusters [49].

Although HPC clouds offer a high degree of scalability, often beyond what is generally available in the general consumer market, they may not be as effective as dedicated HPC clusters due to virtualization overheads and network latencies. Nonetheless, with the high cost of such specialised HPC clusters, continued cloud-based use is expected [5].

### 2.4.2 Services with specific capabilities

Software as a Service (SaaS) technologies offers a limited set of functionalities without the need to install or configure any software. SaaS further abstracts the virtualized infrastructure offered by PaaS, with the underlying platforms managed by a vendor and uptime guarantees usually included as part of the service. These services are generally highly available but may require a subscription-based API to access.

SaaS systems often provide a high level of interoperability with IoT technologies, allowing interaction through application layer protocols like MQTT [50, 38]. [46, 47]

#### 2.4.2.1 Distributed messaging services

Message queuing may be offered as a service, allowing for managed use of cloud-based MOM, ensuring consistent availability. [40] It may also allow direct access to cloud provider-specific features and services.

Large IoT networks with widely distributed devices that need analytics, storage, and supervisory control often use cloud-based message queuing services. MQTT is particularly well supported, but non-commercial use often comes with throughput limitations [50].

#### 2.4.2.2 Specialized applications and processing

SaaS allows for the use of cloud-based functionalities, including highly specialized data processing, modelling and analysis. [7] This is particularly useful when a field relies heavily on software that needs a specific computing environment, e.g. Matlab's Parallel Server (previously Distributed Computing Server). It provides access to pre-configured cloud centre clusters, along with the necessary Matlab packages for parallelized workflows [51].

Full development environments are also becoming available through browser-based online applications. Examples include Google's Colaboratory, which offers cloud-based, GPU accelerated Jupyter notebooks [52], and Matlab Online, which offers a basic Matlab environment with reduced capabilities [53].

### 2.4.3 Interaction with large data sets

Data as a Service (DaaS) decouples the processing and storage of data, allowing for access to large distributed data sets via the internet. DaaS is often seen as SaaS for data and is a relatively new concept, typically considered apart from the traditional cloud service stack [54]. Big data, which involves handling vast quantities of various types of verifiable data, has a strong conceptual relationship to DaaS.

DaaS has found a particular home in the scientific community, where various workflows demand the sharing, viewing, and processing of extremely large data volumes. It necessitates a focus on reduced data transmission bottlenecks for seamless interaction with widely dispersed data over a network.

Due to the potential scope of such data, some processing may be applied to data but still qualify as DaaS overall. Such processing is usually limited to a meta-analysis of the data in its entirety, mostly to provide improved data discovery through subsequent querying and filtering. DaaS is often provided through a custom API, which vendors may restrict in order to facilitate commercialization. It may lead to a measure of vendor lock-in where large relational data sets are in play. There is, however, a concerted effort to make data sets of particular importance freely available through a wider selection of services [3] and allow for standardized access to particular types of data [54].

DaaS facilitates the use of specific data sets by a large number of users. Such interaction may involve crowd-sourcing and live data, leading to frequent changes in these data sets. Open data initiatives, in particular, offer tantalizing opportunities here. [55, 56]

#### 2.4.3.1 Archived sensor data

Some high-level languages can directly access cloud storage services to enable interfacing with remote archived data (e.g. Matlab, which supports integration with Microsoft Azure Storage Blob and the Amazon Simple Storage Service [57]). Data can be pre-processed locally or archived in raw form for server-side processing, with actual implementation dictated by data and system bottlenecks. [16] It is best suited for data that is updated infrequently and polled every few minutes by consumers, such as weather radar data.

A prime example of such a sensor network is the National Oceanic and Atmospheric Administration's (NOAA's) Next-Generation Radar (NEXRAD) Big Data Project. This project allows for the distribution of radar weather data from its entire network of high-resolution S-band Doppler radar stations. Modern cloud storage infrastructures such as AWS and Google Cloud has provided the ideal means to store the large, ever-growing trove of weather radar data. This data is publicly available in CSV format, offering an ideal candidate for testing weather radar systems. [3]

#### 2.4.3.2 Supplementary target information

Additional sensor and telemetry data can provide specific details on targets in the environment. It enables data to be correlated with existing system tracks, resulting in additional information on these tracks. It also allows for tracking performance to be verified, as it typically includes a high-quality position and time of measurement – often provided by a target's onboard GPS unit. Notably, these sources are likely to be out of sync with the radar system, especially when sourced from open-data initiatives.

Ocean-going vessels should be equipped with an Automatic Information System (AIS) unit to enhance safety by improving maritime domain awareness. An AIS unit transmits a unique identifier, along with the vessel's name, cargo type, description, and the relative position of the transmitting antenna. The reporting interval is based on the vessel's speed, but transmit rates may also increase during manoeuvring. [58, 59]

This information can be picked up by local receivers and archived for access through aggregation services [60]. Although real-time data availability is restricted, it is easily available to local maritime authorities [58].

All regulated air platforms should be fitted with ADS-B to ensure compatibility with next-gen ATC systems. An ADS-B unit transmits a unique identifier, along with the platform's state, heading and velocity. The unit reports this information at a rate of 2Hz (far higher than classical airport radars at 0.25Hz) [61].

Nearby aircraft and ground stations may then pick up on this information. As with AIS, there are efforts to preserve ADS-B data for universal access via aggregation services. The update interval may be limited to once every few seconds to reduce overall costs. [62]

#### 2.4.3.3 Geospatial information

Geographical maps are accessible through server-hosted Web Map Service (WMS) providers (e.g. Google Maps and Bing Maps) [63]. The Open Geospatial Consortium's standardised WMS protocol, which enables URL-based queries for geo-referenced imagery from a Geographic Information System (GIS) database, underpins these provider services. [64]

It allows for requesting map tiles at a specified resolution for a geographically defined area. These map tiles are suitable for display on computer screens and provided in a standardized raster image format (e.g. PNG or JPEG). The PNG format allows for the use of transparent areas to create composite views from overlapping map layers. [64]

Numerous open-data initiatives provide map tiles with formatted satellite imagery, elevation maps, and supplemental information (e.g. tiles detailing crowd-sourced listings of transportation infrastructure). This information not only contextualises radar data when presented to an operator but also enables certain tracking functionalities (e.g. line-of-sight calculations and classifying tracks as road-bound vehicles).

Although GIS is a late adopter of cloud technology, it is a good fit due to the prevalence of large data sets and workflows that may require high-level processing [7]. It tends to sit uncomfortably under the DaaS designation as it involves visualizations of data, not the raw data itself. However, common uses fall under the very broad definition of DaaS as SaaS that delivers data, with the exact SaaS/DaaS split frequently determined by the specific implementation [54].

## 2.5 HMI client

This section details the general options for developing the HMI application intended for use alongside the framework, with a broad comparison of both native and web-based environments. This is followed by a rationale for the web-based development stack's selection, along with details on relevant component technologies therein, esp. where considered as essential background knowledge to subsequent chapters.

### 2.5.1 Native and web environments

Native applications are deployed to individual clients, along with all its software dependencies, for stand-alone use. Here these applications are afforded direct API access to device-specific features, allowing for the efficient use of underlying resources. It allows for high-performance, graphics-intensive HMIs, but with a very involved development process.

Native application development often requires the use of a Software Development Kit (SDK). Such may limit cross-platform use or even wholly restrict use to a single platform. These SDKs usually encourage developers to follow a platform-specific design guide, which results in a unified "look and feel" across all native applications. It may reduce design flexibility, as circumventing constraints on control appearance is often challenging.

Web applications are deployed to a host environment from where the necessary logic, resources and dependencies are served to client machines. Here the application runs within a browser-based virtual environment, which allows for indirect access to device-specific features. It results in a performance penalty when compared to native applications, but is still adequate for a variety of purposes.

Various high-level libraries and development frameworks exist that allow for web server and client development. It greatly simplifies the targeting of multiple platforms, without the extraneous effort typically required to achieve the same with native applications. Web applications are also compatible with a wide range of devices, supporting a variety of input mechanisms as well as varying screen sizes, orientations, and resolutions.

Having the application hosted on a server allows for fast access and a single point of interaction for updates. This is a major advantage over native applications, which must be redeployed to each client node. The client usually needs to repeat the site retrieval when the client is relaunched, but for a system using live local data, this may not be a concern.

### 2.5.2 Application stack

Web applications are seen as the best choice here due to their relative ease of creation and widespread availability on current generation computing devices. It should be sufficiently performant for high-level data display, esp. when considering broadly similar SaaS solutions

(e.g. CesiumJS or ESRI's ArcGIS).
It offers some flexibility in system configuration, enabling both the server and client to operate locally or a more comprehensive server deployment in an IaaS/PaaS-based environment for use at scale.

Client applications may be written in various languages, with the most popular JavaScript/ECMAScript (see Appendix A.2.3) or super-scripts that transpile thither. Web server frameworks are quite diverse, with popular implementations in Java, C#, PHP, Ruby, and JavaScript/ECMAScript. [65, p. 73] These frameworks are often suited to a specific purpose and may require specialised domain expertise to apply, with some even restricting the kind of controls that may be readily created.

Node.js (see Appendix A.4.1) may be used for web server development since it enables the use of ECMAScript on both the server and the client [66, p. 7], eliminating the need for a separate server-only programming language. For Single Page Applications (SPAs) [65, p. 76] where the operator is presented with a persisted composite scene, such as a radar HMI, the runtime is relatively performant. With all application resources being requested at startup, the number of client-server interactions is also fairly limited.

Client-side development comprises of a combination of direct HTML5 technologies (see Appendix A.2) and Leaflet (see Appendix A.4.2) to achieve the requisite geospatial views and functionalities. To simplify development, ensure a common build from a varied collection of dependencies and allow for the use of certain current-generation functionalities, WebPack (see Appendix A.4.3) can be used as the application bundler.

The effect of application retrieval on startup time may be largely avoided by implementing the website as a Progressive Web App (PWA), which preserves the site's resources in the browser environment. [67]. Additionally, some settings may be limited to the local client through Local Storage and IndexedDB, reducing the need for interaction with server-side data structures to realize certain functionalities.

## 2.6 Human-Computer Interaction

This section details some of the visual design aspects for a modern operator interface, with particular regard to core principles and the broad HCI trends premised thereon. Considering the radar system as a whole and not simply a sensor, it is clear that for an effective tool allowing purposed usage (as in section 2.1) the operator is a crucial part of the system. Unfortunately, this factor is sometimes neglected or even wholly overlooked with broader considerations to radar system development.

As discussed in section 2.2, modern radar systems are comprised of various subsystems, each of which serves a distinct purpose. In general, defining intra-system communication necessitates a thorough examination of various trade-offs in the context of a particular environment and usage. This involves anticipating and removing inefficiencies that might lead to bottlenecks in specific scenarios. By considering the operator as another subsystem in the wider radar system, the same considerations apply as with any other subsystem.

With its emphasis on the empirical study of operator interaction with system components, Human-Factors Engineering (HFE) has made significant contributions in this area. HFE enables HMI designs to be tailored to a target population's capabilities, taking into account anatomical proportions and range of motion, as well as the impact of ambient lighting conditions, vibration, and high-stress situations on performance. [11]

Iterative advances and research have resulted in a body of work that formalises efficient means of operator engagement with computer systems. The military domain, in particular, has seen the adaptation of various best practices [10, 13], as any lapse in quality can lead to a decrease in task performance and safety. [68]

The need for improved perception, cognition, and reaction times have resulted in a trend favouring uncluttered designs and controls with low visual complexity. Advancements in computer and interface technologies have especially benefited cockpit design, enabling sustained operator awareness of the plethora of systems aboard modern aircraft. [68, 12] The same underlying principles that govern cockpit design have found their way into a wide range of operator-controlled systems [13], often as a result of these exhibiting the same rise in complexity as with aircraft.

To avoid confusing or surprising the operator, it is also crucial that the same design

language be used throughout an HMI. This consistent approach ensures that the operator builds a level of trust in the system and its operation, particularly when it proves to reduce discomfort and workload. Extending a design across HMIs also allows for faster operator acclimatization to new systems, as training on one system is transferable to similar systems.

### 2.6.1 Basic considerations

Optimal information transfer requires that the operator's focus be carefully managed with specific visual stimuli. A good design should consider the limitations of human vision, using luminosity, colour, contrast, and scale to convey meaning and facilitate decision making.

Ambient lighting may make HMI use challenging, particularly in direct sunlight or dark rooms. A light background with dark markings is broadly recommended as it reduces screen glare and improves legibility. A dark background, on the other hand, is desirable when night vision must be preserved, colour differentiation is crucial, or eye strain is likely due to prolonged use in low-light conditions. [69]. Contemporary systems often provide a light/dark mode toggle where a range of usage environments is likely. [68, p. 12]

Colour is most effective when used sparingly, acting as a supplementary form of coding to shape for enhanced recognizability. [13, p. 152] Since primary colours are easier to distinguish, red, green, orange/yellow, white, and blue are typically employed. [70]. Warning and caution states are usually indicated with saturated red and orange/yellow, but this may vary based on cultural factors. [17] Insensitivities to specific frequencies of light may reduce the efficacy of colour coding. The incidence is population-specific, but it can be quite high, with up to 8% of males affected in some populations [71], this demands a careful selection of hues and the use of luminosity as a secondary form of visual coding.

The readability of text on digital screens presents unique challenges. Contrast affects legibility significantly, with dark text on light backgrounds easier to read than light text over dark. Where certain usage considerations require light text to be used on a dark background, glyphs should be elongated by about 25 to 33% to avoid visual bleeding effects [10, p. 135]. Sans-serif font types are suggested for smaller fonts or lower resolution displays [68, p. 51] as it tends to be rendered more crisply.

Elements need to be of sufficient size for accurate shape recognition. Sizing is expressed as

a visual angle to account for differences in the distance at which displays, such as mobile phones and projected screens, are expected to be used [11] and expressed by:

$$\text{Visual Angle (Min.)} = 57.3 \cdot 60 \cdot \frac{L}{D} \tag{2.1}$$

Where $L$ is the object size and $D$ is the eye's distance from the object. [13, p. 192]

Complex shapes, such as symbology, should subtend at least 20 minutes of arc [13] assuming normal visual acuity. Alphanumeric characters are indistinguishable under 5 minutes of arc [11, p. 941], with common guidelines well in excess for NRT systems. Glyphs should subtend a minimum of 15 minutes of arc but preferably fall in the 20-22 minutes of arc range for normal text [13, p. 58] and 24 minutes of arc for critical controls [68].

#### 2.6.1.1 Optimizing for human decision making

After being provided with information, the operator needs time to contextualise, decide on a response, and then, if any action is needed, find the appropriate controls that carry out the decision. It is broadly analogous to computational decision-making systems, with the fundamental principles governing efficient HCI being drawn from information theory.

HMI sections should be meaningfully organized into distinct sections to aid with cognitive comprehension [72]. The operator can further be assisted in contextualizing information through the use of size, shape, colour, pattern [18] and blink coding. This provides designers with the means to direct operator attention and should be used with due consideration. It is particularly pertinent with NRT systems, where the use of unique symbology can allow for fast cognition and task prioritisation under high stress conditions. [13, p. 149-164]

The HMI should not flood an operator with information, as there is a nebulous but definite limit in the human ability to visually process and interpret data. This can be mitigated by limiting information based on task applicability [13, p. 30], grouping information by type, intelligently fusing data [12, p. 12] and reducing short-term memory load (e.g. fragmenting long workflows into limited interactive sequences). [13, p. 107-110][11, p. 178-254]

Although operator experience improves HMI usage performance to some extent, the time it takes an operator to execute decisions through a specific design may still be roughly

predicted. The basic principle is described in the Hick-Hyman law, which approximates basic selection time when presented with clear visual stimuli in a list through:

$$T = a + \ b \cdot \log_2(C) \tag{2.2}$$

Where $T$ is the time to choose an item from $C$ equally probable items, with $a$ and $b$ empirically derived constants. [73, p. 2]

This shows a logarithmic rise in the time to make a decision when directly presented with a list of options, which typically results in a practical limit of 10 items in a list (with 3-5 elements being preferred) [13, p. 120]. [73] As such, nesting items into meaningful groups is beneficial when presenting a large number of elements.

#### 2.6.1.2 Ease of interaction

Anthropometry, the study of human body lengths and circumferences, is used extensively in guidelines stipulating the general environment within which the HCI is used. [10] General operator physical capabilities and range of motion inform design decisions for common or critical tasks. Although this mostly concerns the physical HMI console, it does pertain to some aspects of HCI design, such as button size and placement. [11, p. 38-93]

The basic difficulty in executing a particular action is expressed in Fitts' law, a fundamental predictor in human motor behaviour [74]. It represents an estimate of the execution time that occurs after the operator has interpreted visual stimuli, decided on a response, and formulated actions, with:

$$T = a + \ b \cdot \log_2\left(\frac{A}{W} + 1\right) \tag{2.3}$$

Where $T$ is time to effect the action, $A$ is the distance from the starting point to the center of the targeted control, $W$ is the effective target width, with $a$ and $b$ specific constants related to the efficiency of the pointing system. [74]

Fitts' law has been modified extensively to account for various factors, notably screen edges and touch screen input. However, the core premise remains that actions requiring the use

of small elements located far away take longer to execute than actions to large nearby elements. [74]

Guidelines generally stipulate button sizes on touch displays in the range of 9.5-22mm, depending on usage, with a dead space of at least 3.2mm between buttons. Using the higher values in size ranges is advisable with vibrating environments or gloved use. These introduce a measure of unpredictability, with the increased sizes then compensating to ensure intentional actuation at all times [13, p. 66]. [10, 11]

### 2.6.2 Trends in the industrial sector

The industrial sector has strict guidelines on various HCI design aspects due to safety considerations and the potential cost of inefficiencies. [11, p. 900-1093] Military HMIs, in particular, tend to focus on role-defined usage and may have specific requirements due to their usage contexts.

HMI operators may be vetted to a specific baseline, e.g. full colour sensitivity where colour coding is used. Certain conditions may necessitate a specific design focus to ensure safe usage, with no single design able to provide a generally acceptable solution. Factors that affect HMI design include high-stress use, mobility, severe vibration, temperature extremes, and the use of specialised equipment. [13, p.30-32]

Although acronyms and abbreviations are typically discouraged, they enhance clarity when formal training is taken into account – especially for military and aerospace use. Mixed case lettering emphasises these acronyms and improves legibility. [68, p. 51, 53]

Pseudo-three-dimensional displays are frequently proposed for HMI displays. It is, however, not appropriate here as it distorts perspective, increases clutter, and reduces operator cognition. [70, p.785] Even when an additional dimension of information must be visually examined, profile views often provide more emphasis and clarity. [18, fig. 2.2]
As such, a top-down, 2D tactical map-driven view with mouse and touch-based control is advisable. It is exhibited in various modern designs in the civilian (FAA NextGen ATC), scientific (NASA HFE [17]), and military (missile defence [21]) domains.

A basic map-based interface should provide the operator with a clear sense of their overall location and, ideally, allow for continuous panning and zooming, as well as a quick action button to re-center on a specific home location. Map elements should be presented as visual layers that can be faded out or hidden. Visual clutter should be minimized, with detail information kept to contextual popups. The layout should be designed such that critical elements are not obscured when an operator reaches across to activate controls. A declutter button should also be available to quickly hide non-critical information. [13]

HMIs in the industrial sector also see a strong emphasis on symbol and colour coding, essentially creating a domain-specific visual language that aids with situational awareness and specific operator duties.

Perceptual layering simplifies the addition of supplemental visualisations, such as weather data and range rings [17]. It provides operators with a selective display of features as well as some control over the visual hierarchy. [16] Attention management techniques such as high luminance-contrast variance can be used to highlight primary data. Tracks are placed at higher perceptual layers to avoid obscuration [68] and make use of high saturation colours to effect a measure of popout [17, phase 2]. Track bearing is often indicated symbolically, with a label containing a unique identifier along with high-level information such as track speed. [70, p. 30]. [17]

### 2.6.3 Trends in the consumer sector

The increased availability of computing systems to the general public has resulted in efforts to make their operation broadly accessible. Even where a system design is focused on military or industrial use, it is still beneficial to take consumer sector trends into account. This helps to reduce operator training time by allowing for some skills transfer.

Platforms usually specify a visual design language that applications are expected to follow. These languages codify the layout, colour, size, and typography to use. With consumer HMIs a greater focus is placed on aesthetics, while still broadly adhering to the same basic principles as industrial HMIs. There is also an awareness of general User Experience (UX) as an overarching concept distinct from individual User Interface (UI) design, emphasizing user satisfaction and comprehension.

Early design languages heavily featured skeuomorphic design elements to provide a sense of familiarity to untrained users when introducing new features. As consumer access to technology grew, more abstract flat designs became the norm, particularly on the web and mobile devices. Its minimalist styling reduces clutter and allows for a more interactive approach to information display. [72]

Control and data components are more distinct in flat designs and work well with layered maps. Google's Material design is particularly noteworthy, as it is premised on the use of shadows to create elevated regions [75], resulting in an improved contrast ratio. It is also familiar to a large segment of the general population due to its broad adoption in many modern web designs. This not only reduces the need for formal system training but also enables the use of a mature design language tailored for highly adaptable interfaces.

# Chapter 3

# Methodology

This chapter outlines processes to investigate the suitability of the technologies under consideration for high-level radar system development. It broadly adopts an experimental approach, exploring specific aspects of the proposal from a developed implementation.

The broad study of literature pertaining to the use of protocols, especially wrt higher-level messaging (see section 2.3.4.2), is insufficient for implementation of the application, necessitating further investigation. A preliminary study aims to fill this gap, emphasising aspects as specifically required for subsequent development.

With sufficient familiarization to the characteristics of the messaging protocols of concern, an architecture for the implementation is defined. This architecture includes a definition of the virtualized subsystem blocks that fundamentally describe the framework, as well as the logical and visual components that associate thereto.

Implemented subsystem modules are then used alongside the HMI application to assess the viability of the proposal for common high-level radar usage scenarios. To this end, various signal chains must be dynamically constructed, with their verification emphasising certain usability expectations. This includes high degrees of flexibility and modularity given the continued differentiation and rapid evolution within radar systems development, enabling piecemeal upgrades in both on-premises and cloud-based environments. The full-chain throughput performance for a selection of data types can then be measured using the implemented signal chains, determining the broad suitability of their use.

## 3.1 Preparatory investigation

A preliminary study investigates particular aspects necessary for the circumspect design of a prototype implementation. The study is intended to provide an early indication of the appropriate use and applicability of the protocols under consideration and assumed as part of the risk handling aspect of the spiral model of the software development life cycle.

It details a more in-depth investigation of the characteristics of the protocols identified in section 2.3.4 for potential use in the framework (i.e. TCP, UDP and MQTT). As such, it expands on Cui's MQTT findings [41] with a particular focus on the factors of most concern for implementation, such as the use of MQTT/WebSockets, higher performing general purpose hardware, and pure UDP or TCP for high throughput data transfer.

## 3.2 System design and implementation

Having characterised the protocol aspects of particular note, it is now feasible to implement a high-level architecture for the client and associated subsystems based on the advantages of the identified protocols, particularly the MQTT IoT protocol. It entails a design that leverages the broker's extensive data model structures rather than the tightly linked exchanges of traditional client-server systems.

The fundamental premise is to dynamically construct a signal chain from a collection of self-describing subsystems, with the help of an HMI that enables operation and integration of these subsystems into single-flow signal chains, as shown in figure 3.1.

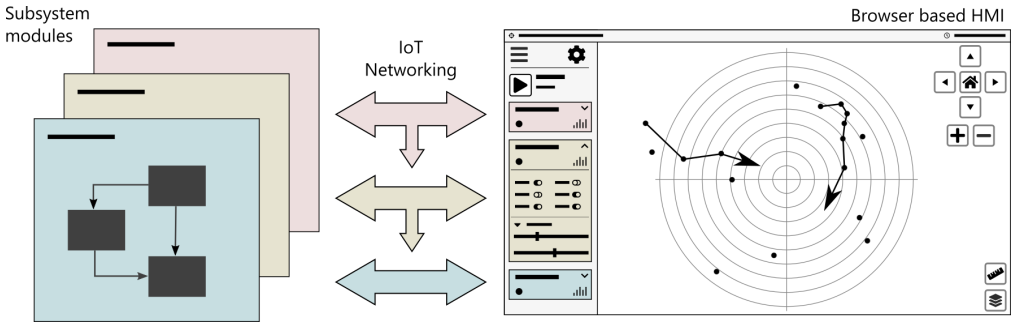


Fig. 3.1. Conceptual diagram of the proposed system to its associated HMI

To enable a system that is dynamically assembled from independently developed subsystem modules, a broad architecture outlining the core system components is defined. It must provide a suitable blend of the Pipe & Filter architecture with the Hexagonal architecture as and where it ensures specific benefits for an adaptable radar system framework.

To ensure a design that is receptive to the use of various subsystem types, even those not detailed in specific, these modules make extensive use of flexible data formats for interfacing. It gives the subsystem developer the necessary flexibility to define specific parameters as required to a significant degree.

The visual design for the related browser-based HMI is then presented through a series of Inkscape sketches, taking the essential factors and best practices as outlined in section 2.6 into account. Although the design itself is not considered a focus of testing, it should allow for reasonable operator use of the system and its features. It should support both mouse-based desktop and touch-based mobile device usage, ensure the ongoing monitoring of an active signal chain and its modules, and facilitate data-driven workflows.

## 3.3 Usability verification

The previously defined framework allows for a sample of system implementations to be evaluated for basic functionality and performance. Specific cases are selected to demonstrate general principles of use that may be expected to apply to a broader context.

Mainstream resources and tools such as generally available hardware and web services constrain these evaluations. A measure of subjectivity is assumed in identifying the point at which system utilisation becomes severely impacted. Here the basic usability, rather than usage experience, is used to establish the limits of HMI responsiveness.

The browser-based HMI's overall operation may then be verified, including its ability to adapt to desktop and mobile device displays and conformance to the broad HCI factors for safe use in various operator environments.

A selection of representative signal chains allows for assessing the viability of on-premises implementations. At first simple single element chains is used to demonstrate very basic functionality such as direct viewing of a simple high-level data feed (e.g. AIS, ADS-B,

or NEXRAD weather data), interop control of co-located systems, and data recording. Thereafter, multi-level signal chains shows functionality such as data playback and fully actualized tracking of targets from emulated data. With the basic operation determined, a simple cloud-based (e.g. AWS, Azure, or Google Cloud) deployment is then considered.

## 3.4 Performance characteristics

Finally, a series of tests are used to establish the performance characteristics of specific high-level data elements. These not only encompass network throughput and latency of various payloads but the full implementation stack, including the interpretation and display of data, to identify the primary bottlenecks of practical concern.

Measurements employ custom HMI indicators and other in-browser tools. Such a setup should be sufficiently analogous to a fully realised system, with its inherent impacts and potential bottlenecks, to show the actual utilisation constraints.

The principal case for data transmissions here is a continuous 4-second scanning radar, with some consideration given to a faster 1-second tracking radar where applicable. This applicability is determined by the extent of dynamic structures that increases complexity and memory use.

Latency and throughput limits are determined where display elements fall out of sync with the system, resulting in timeouts and removal of those elements. HMI limits are decided by the application responsiveness as indicated with continuous processing maxima on the same processes that handle operator inputs (e.g. panning and zooming).

Specific tests include the system startup times and the functional limits for high-level radar data such as clutter, plots, strobes, and tracks. Output testing focuses on continuous transfer rates and peak behaviour with data bursts that the system may recover from for local and cloud-based subsystems.

These provide a benchmark to evaluate the suitability of the technologies under consideration to specific requirements in various radar systems and deployment scenarios. With the defined analyses allowing for a broad determination of usage characteristics, the general trends outlined may be considered universally applicable and transferable.

# Chapter 4

# Preliminary Study

This chapter details the first step in the outlined approach (see section 3.1) to ensure the viability of the subsequent design. Here the previously selected (see section 2.3.4) transport and application-level protocols are experimentally analysed and compared in terms of their suitability for specific data and control message transfer operations.

The specific protocols under consideration include UDP and TCP (see section 2.3.4.1), as well as MQTT/TCP (see section 2.3.4.2). The referenced literature emphasises protocol usage on low-performance devices. More testing is required to determine usage characteristics and effective throughput on mainstream hardware. These tests allow further limiting of the protocols under consideration and also inform subsequent system design.

Considering the use of a web application (see section 2.5.2), the protocols discussed in section 2.3 can not be considered as sufficient. Protocols need to be layered over either HTTP or WebSockets to allow for efficient browser interaction (see Appendix A.3) with web applications. WebSockets is preferred here due to allowing for event-based interaction and many MQTT brokers directly providing WebSockets support [44].

The MQTT broker used in the solution, mosquitto, allows both pure and WebSockets based communication, albeit on different ports. MQTT/WebSockets supports all QoS levels (i.e. QoS0, 1, and 2), which are evaluated individually in this assessment. The impact of Secure WebSockets (WSS) use is not evaluated here, as the test setup makes it challenging to isolate WSS-specific transfer behaviours from broader network influences.

## 4.1 Protocol tests

Specific tests examine the characteristics with a range of payload sizes. These include typical usages such as single message transfers (e.g. 26 B and 1400 B) and compressed payloads (e.g. 16 kB and 64 kB) for a higher payload to header ratio.

Results are indicated graphically for different payload sizes, with the pass-through failure ratio, mean latency, and mean data throughput considered in turn. These metrics span the period from the creation and enqueuing of the message at the sender to its dequeuing at the recipient.

The performance of each payload/protocol combination at different throughput loads is then considered by queuing up several transfers in rapid succession. These queue sizes are measured in powers of ten, ranging from one, which represents an infrequent message or a densely packed collection, to ten thousand, which represents system behaviour with continuous data transfer. The mean of five consecutive samples is taken, with an error bar showing sample variance.

The following subsections depict the transfer characteristics of the payload sizes under consideration. These can predict transfer characteristics and throughput in a system when the nature of that system's transfers are understood. Given that the most appropriate protocol may depend not only on the anticipated network inefficiencies but also on the available system resources, any observation should be regarded alongside the details provided in section 2.3.

Tests were performed on a Dell G5-5590 laptop with an Intel Core i7-9750H processor and 16 GB of RAM, with all samples taken over a loopback and the network-specific configurations at the Windows 10 defaults. Python 3.8 was used to write the test code, which enables semi-automated testing and the generation of reports that detail the inter-process transfer characteristics of samples.

### 4.1.1 26 B payloads

Figure 4.1 depicts a protocol comparison for messages with 26 B payload sizes – analogous to basic command messages or rapid updating counters, such as boresight indicators.

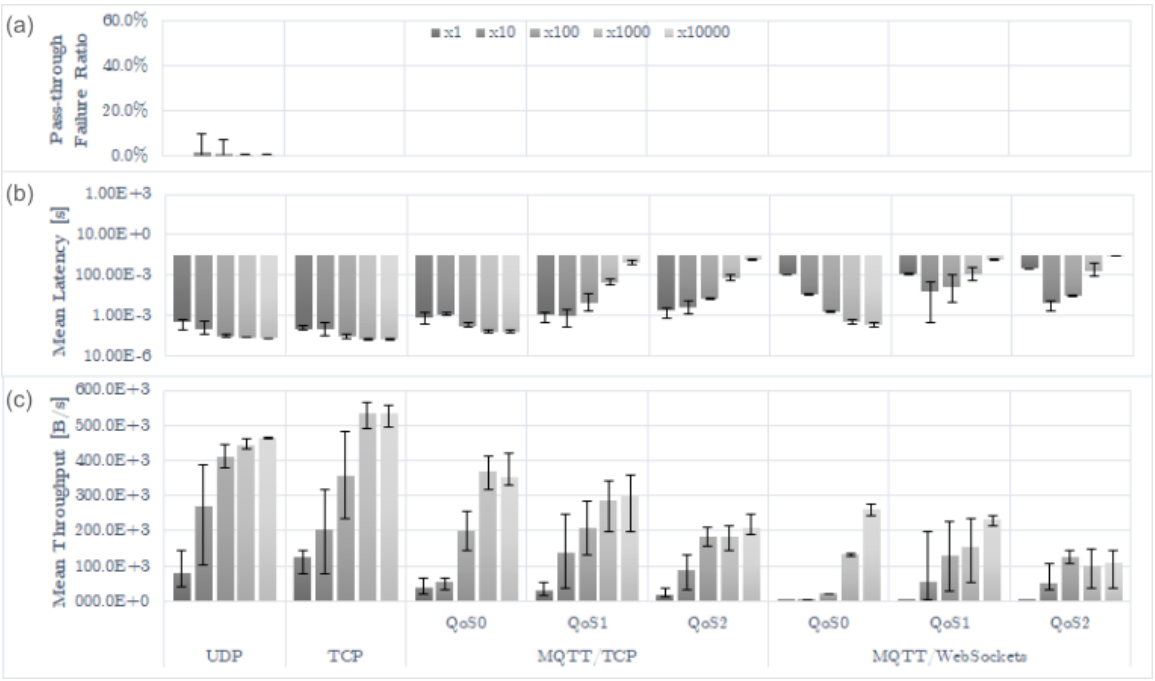


Fig. 4.1. Comparison of 26 B payload size protocol performance

a. UDP exhibits pass-through failures where a small number (although not singular) of messages are enqueued for transfer in rapid succession. Failures are reduced with a large number of queued messages, with sub-percentage values for continuous messaging.

b. As the number of enqueued messages rise, the mean latencies for the base configurations fall. Continuous transfers show a low latency variance, implying that inefficiencies are eventually eliminated. The opposite occurs with higher QoS levels on either of the MQTT protocols, with latencies rising noticeably instead. This is most likely due to retransmissions resulting from transfer validation or the use of a delay to cope with a large backlog of unconfirmed transfers.

c. Throughput is low with all protocols, likely due to inefficiencies from the high overhead for such a small payload size. MQTT/WebSockets QoS2 outperforms QoS0 for low queue counts, indicating that message spacing may increase throughput.

TCP looks to be the best choice for continuous small payload transfers, owing to transfer-level packing, which improves the header-to-payload ratio.

### 4.1.2 1400 B payloads

Figure 4.2 depicts a protocol comparison for messages with a 1.4 kB payload – analogous to small numbers of packed plot data or even large track messages. Using this message size also prevents fragmentation by older network routers (depending on protocol header size).

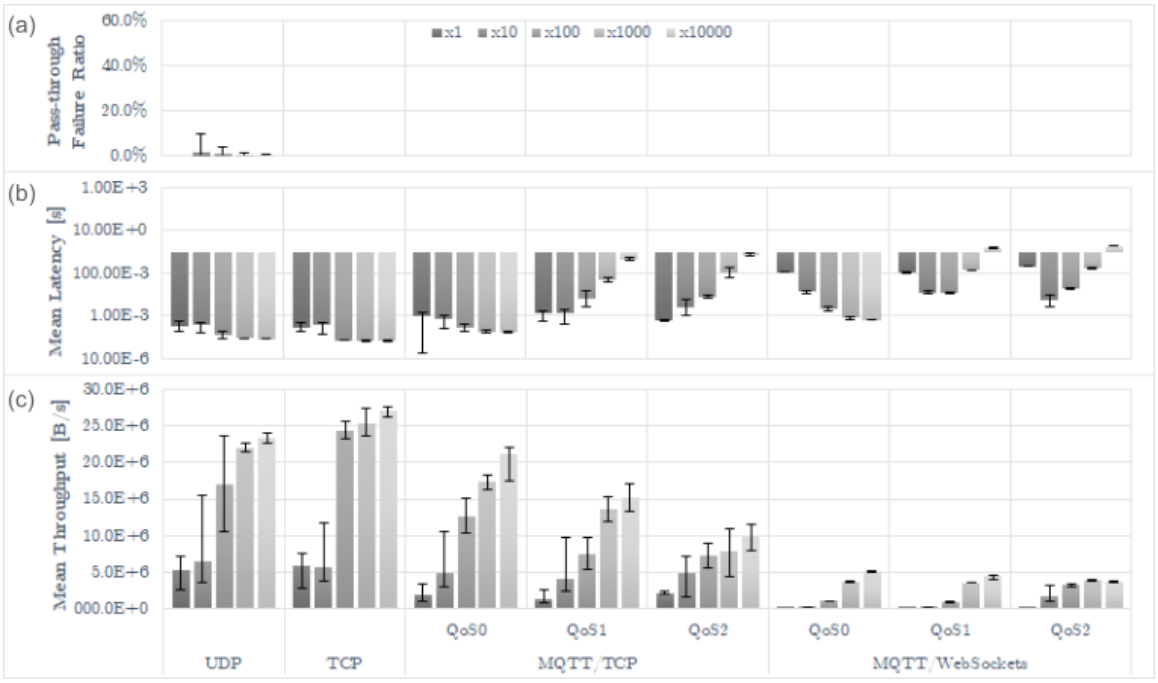


Fig. 4.2. Comparison of 1.4 kB payload size protocol performance

a. This payload size is sufficiently small that it has similar pass-through failure ratio characteristics to a 26 B payload (see 4.1.1).

b. Mean latency characteristics are the same as with a 26 B payload size (see 4.1.1).

c. Throughput scales near linearly to the payload size, with the near 50-fold increase in payload size resulting in a roughly 50-fold increase in the mean throughput.

UDP nears TCP's throughput performance here at times, but TCP remains preferable, esp. considering the high risk of pass-through failures with UDP.

### 4.1.3 16 kB payloads

Figure 4.3 depicts a protocol comparison for messages with 16 kB payload sizes – analogous to larger sets of packed data, such as plot or track collections, or even raster image sectors.

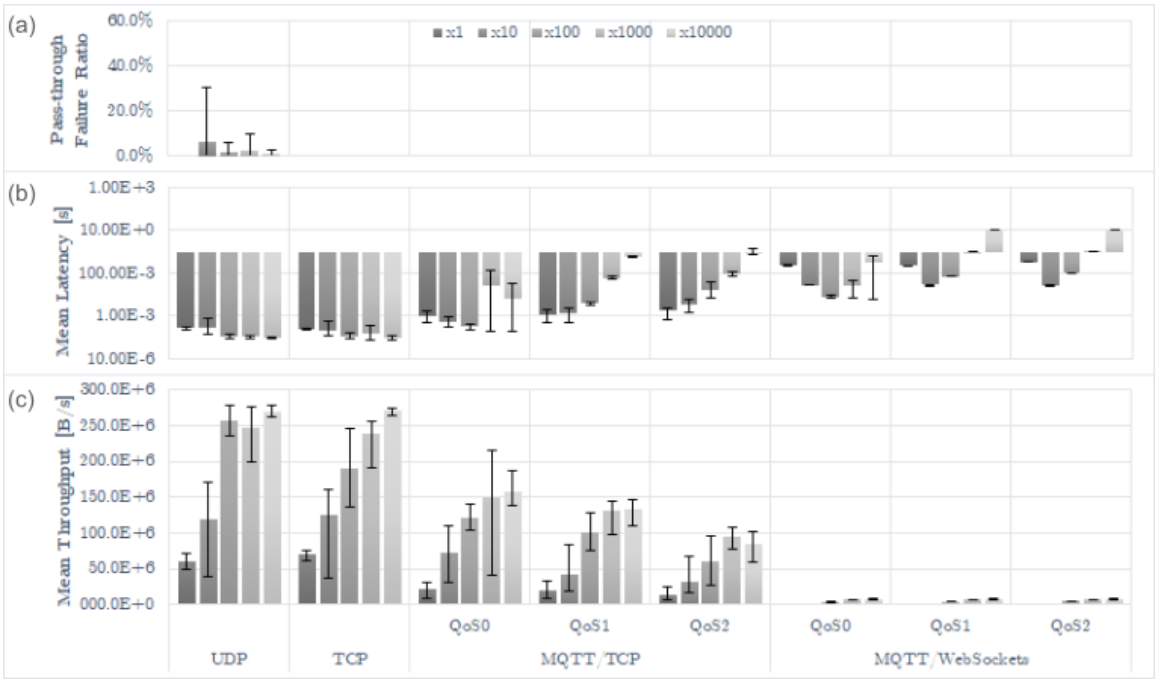


Fig. 4.3. Comparison of 16 kB payload size protocol performance

a. Overall, UDP pass-through failure ratios are higher here, yet all single-message transfer samples still succeed without incidence. This indicates that spacing successive messages may actually improve overall throughput with UDP.

b. With continuous throughput, MQTT QoS1 and QoS2 show excessive latencies. When using these configurations, messages should occur infrequently or have a small payload.

c. MQTT/WebSockets throughput previously scaled with payload size; here, however, continuous transfers are clamped at around 7-8 MB/s, with throughput still scaling as expected with the other protocols. TCP throughput fluctuates greatly, although it stabilises with continuous message transfer.

TCP is still considered preferable, especially given UDP's higher pass-through failure ratios without significantly outperforming TCP.

### 4.1.4 64 kB payloads

Figure 4.4 depicts a protocol comparison for messages with 64 kB payload sizes – analogous to data files, larger image sectors, or even full-size radar maps for a low-performance system.


Fig. 4.4. Comparison of 64 kB payload size protocol performance

a. UDP shows high pass-through failure ratios that rise with queue sizes, likely due to default system settings limiting buffers more than on a dedicated environment. Caution is required, as failure rates may exceed what statistical algorithms can account for.

b. TCP latency no longer reduces with continuous transfers but increases due to congestion. With a continuous transfer of enqueued messages, MQTT shows high latencies for all QoS levels. This aligns with Cui's observations for MQTT on Raspberry Pi devices, where latency jumps were seen with payloads larger than 16 KiB [41, p. 42-45].

c. UDP throughput is quite high, sometimes exceeding 1 GB/s. TCP throughput is much lower than the other protocols shown here, most likely due to fragmentation during encapsulation. MQTT/TCP is worth considering for large data bursts, as it outperforms standard TCP without the packet losses associated with UDP. This is most likely due to application-layer control over packet sizes, which also accounts their higher latencies.

UDP shows very high throughput relative to the other protocols, but with its high pass-through failure ratios, message queueing and network settings must be considered with care. MQTT/TCP use is also viable where messages are not transferring continuously, allowing buffers to clear.

## 4.2 Narrowed protocol selection and considerations

As MQTT/WebSockets enables push-based communications with a browser-based client, it is given the highest weight in this category. Since MQTT/WebSockets has a lower throughput and a higher latency than the other protocol options, the solution should include TCP/IP as an alternative data transfer method (esp. given excessive UDP losses).

Due to MQTT/WebSocket's proclivity for very high latency with large messages and limited throughput with smaller payloads, processed data may be prepared as packed payloads of up to 16 kiB in size. Each MQTT message can then contain up to 256 plots (assuming a 64 B plot size) or 64 tracks (assuming a 256 B track size).

Figure 4.5 magnifies the MQTT/WebSockets QoS comparison from figure 4.3. It shows that protocol throughput is effectively limited to about 7-8 MB/s, with similar behaviour as shown with a 64 kB payload size in figure 4.4. This is unexpected considering [76], and seems to indicate a broker implementation constraint or queue size restriction by the MQTT library.

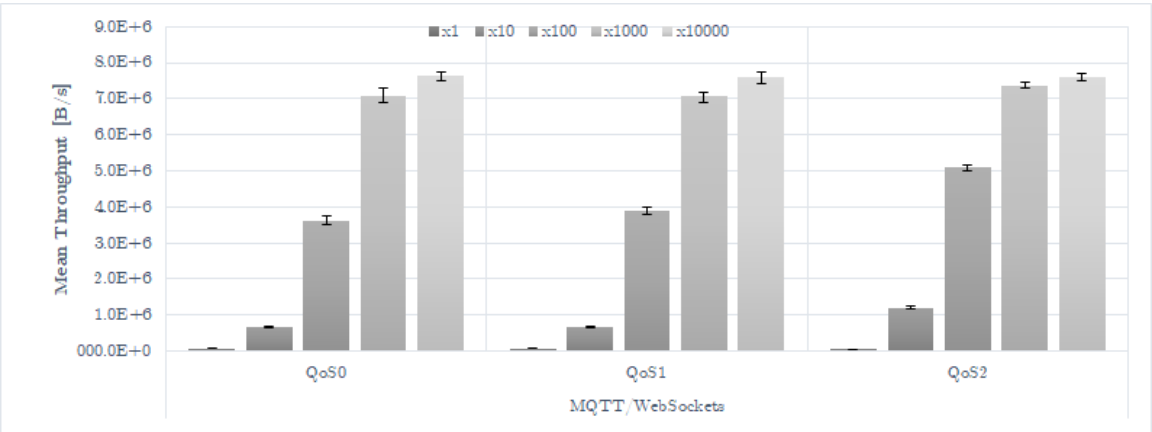


Fig. 4.5. Throughput performance with MQTT/WebSockets using 16 kB payloads

QoS0 is chosen for data transfers because it has significantly lower latencies when data is continuously transferred (see figure 4.3), despite having slightly lower throughput than QoS2 in some of the scenarios considered (see figure 4.5). With a mean continuous upper limit of around 7 MB/s assumed, up to 427 payloads (16 kiB each) can be transferred per second. Solely based on protocol constraints, this results in an upper limit of approximately 109 thousand plots or 27 thousand tracks per second.

To contextualise this with regards to the uses described in section 2.1: a low-performance 4 second rotation time scanning radar with 65 568 clutter points (32 B each), 4 096 plots (64 B each), 64 strobes (64 B each), and 256 tracks (256 B each) per rotation requires a transfer rate of 608 kB/s. As such, the MQTT/WebSockets protocol limits considerably exceed the rates required for viable high-level radar system use.

It is important to note that these limitations only apply to data intended for browser-based use. Low-level data (such as that of USRP2, as indicated in section 2.3.1) may be sent at much higher throughput rates and with lower latencies between subsystems by using any of the other protocols detailed in this section.

In considering the established performance constraints and broad characteristics of MQTT/WebSockets, a radar system framework exploring its use should allow for alternative high-throughput interfaces. These are likely to necessitate more direct supervision by the subsystem developer but may be assumed as limited to data transfers where the HMI is not anticipated as a direct consumer. With this in mind, it is now feasible to proceed with an implementation that makes use of this representative IoT protocol.

# Chapter 5

# Implementation

This chapter details the actual implementation and working of the refined design. It presents the architecture and broad entities within a radar system using the proposed framework, followed by a visual design for the associated browser-based HMI.

## 5.1 System architecture

This section provides an overview and design of the system and its parts, allowing for a Signal Chain architecture that is specifically adapted for IoT use.

Interconnecting components should be limited to a bare minimum for simplicity of deployment. Intercommunication is intended to take place via an MQTT broker that supports WebSockets (see section 4.2), which enables publish-and-subscribe messaging to the majority of modern web browsers [77]. The broker-hosted JSON data model, which provides a detailed description of the system (see Appendix C), is central to the versatile multi-tenancy intended here.

The complete data model can be divided into smaller topic-related sections, enabling message filtering and subsystem targeting. Elements in the data model are identified through Globally Unique Identifiers (GUIDs), allowing for user-friendly element names that may be modified without breaking the system.

Since any related subsystem or connecting client can modify the data model, all associated components of the configured system must be responsive to changes and adapt dynamically. Data transfers are in the form of UTF-8 encoded CSV records to simplify interoperation and debugging, with a separate associated topic containing the headers that describe them.

Figure 5.1 depicts the proposed system's conceptual design and high level data flows. Subsequent sections provide more detail on the client (see section 5.1.1), broker (see section 5.1.2), Signal Chain component configurations (see section 5.1.3) and the general subsystem template that may be used to create these configurations (see section 5.1.4).

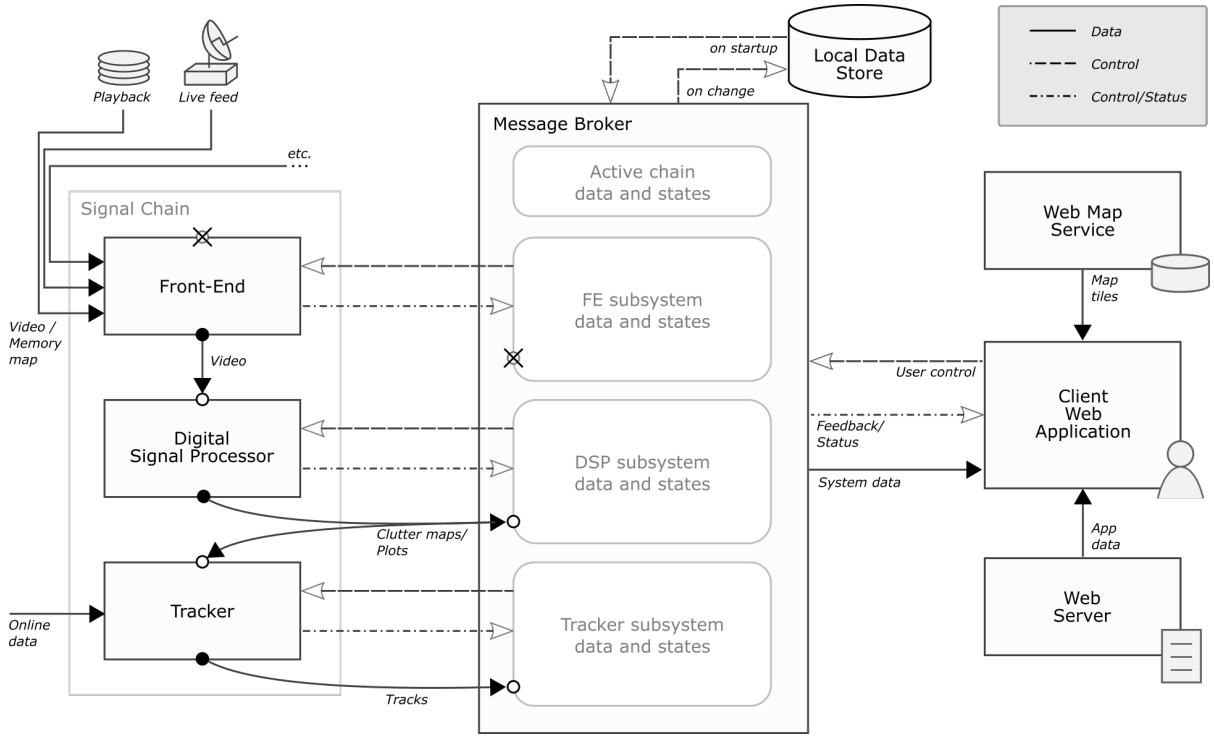


Fig. 5.1. General representative block diagram and data flow of proposed system

### 5.1.1 Client web application

The HMI comprises a simple SPA that can leverage PWA technologies for quasi-standalone operation on modern browsers that support it (see section 2.5.2). An HTML web server distributes the requisite files for deploying the HMI to connecting browser clients. This web server may be co-located with the central broker or hosted on a cloud platform.

After retrieving the necessary resources, the application may provide system functionalities without subsequent interaction with the web server.

Map tiles are fetched directly by the application from online sources in the default configuration, requiring client-side internet access. With WMS based on a common standard (see section 2.4.3.3), minor adaptations to the basic implementation should suffice if a custom local WMS is desired.

Within the data model, some structures can relationally provide information about other elements. It allows for selectively defining interlinked subsystems, such as shown in figure 5.1, by instructing clients on which topics they should subscribe. Using brokers with WebSockets support enables event-based multi-tenancy, ensuring that all connected clients and subsystems stay current with any changes made to the data model.

A high-level data-driven design is enabled by limiting the available data and control types. These are defined in structures nested under their parent subsystems for greater modularity. Defining strictly codified structures in the data model allows for generalist logic to interface with these structures as appropriate.

Figure 5.2 provides a typical activity diagram for interpretation of the subsystem's output data schema. Each schema describes the fields and their specific order in the associated record, as for a particular type (e.g. tracks, plots, strobes and heat maps).

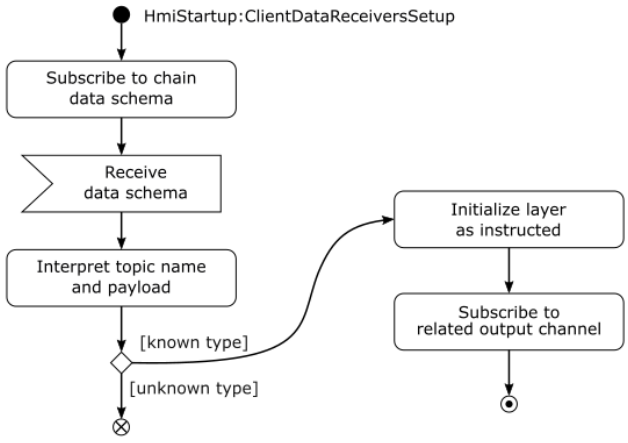


Fig. 5.2. Activity diagram for client-side data interpreter setup from schema topics

Controls may also be dynamically placed on the HMI to depict relationships defined within the data model. The client looks for codified structures describing general controls, such

as sliders or radio buttons, for inclusion in the HMI. Selective monitoring of changes to these structures ensures that synchronization is maintained with controls.

Figure 5.3 provides a typical activity diagram for interpretation of the subsystem control model. Here the message serves as a schema describing new subsystem controls, as well as a data model that updates existing control values.

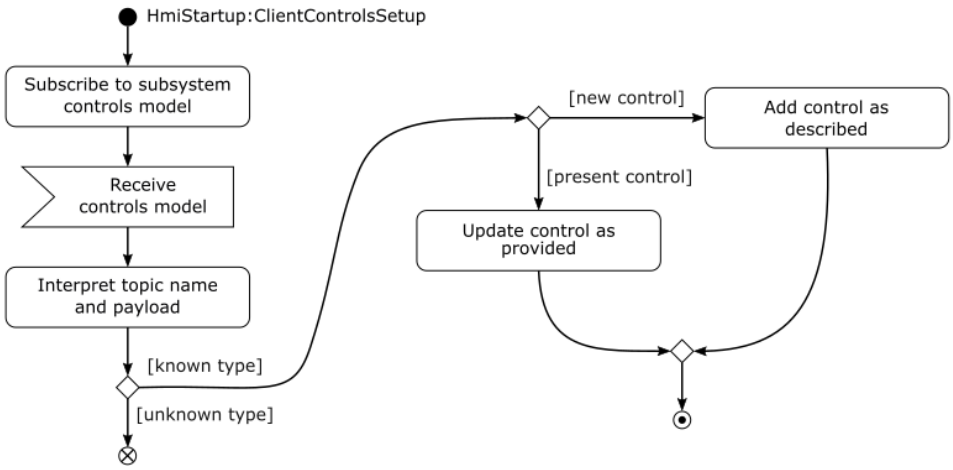


Fig. 5.3. Activity diagram for client-side controls from schema topics

Some settings are kept local to the client and preserved across sessions by using the browser's IndexedDB. These settings include connection details to the central broker as well as client-specific style options.

### 5.1.2 Message broker

The centralised infrastructure holds the data model and stores system states and data for on-demand access, as shown in figure 5.1. By restricting broker access to approved clients, access to the subsystems configured to the broker may also be limited.

Due to the lack of typical server architecture, connected entities must provide all the necessary data model management functions through their section in the data model. The broker also provides a centralized backup mechanism that periodically mirrors the control topics to local file storage. This allows configurations to be retrieved during the broker's initialisation process, reloading messages with LWT set as the broker host starts up.

### 5.1.3 Signal Chain

The central broker's data model defines the connections between subsystems. It allows these nodes to be sequentially joined into a distributed signal chain, as shown in figure 5.1, to accomplish specific outcomes. The implementation is simplified by only allowing one signal chain to be active at any given time; however, several chains can be stored within a data model and subsequently selected.

Depending on the preferred data transfer profile (see chapter 4), interconnections can be configured as either a straight TCP/IP stream or MQTT pub/sub. Web clients are intended to enable directly tapping into an interconnection for display purposes where it uses MQTT. While a subsystem may have multiple data outputs, only one of these may be used by the following subsystem, with the remaining generally going to the client for display. Hereby the individual subsystem nodes may be categorised into four fundamental types by their use of provided I/O data channels, as seen in figure 5.4. The specific configurations are subsequently discussed.

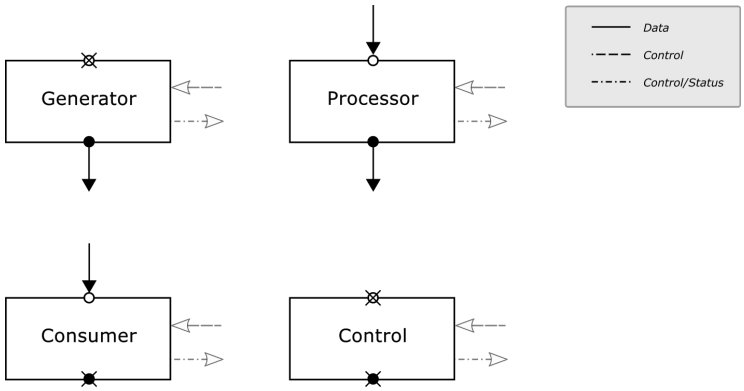


Fig. 5.4. Subsystem module configuration types by input/output data channels

#### 5.1.3.1 Generator configuration

This configuration is analogous to the classic Generator block (see section 2.2.2.1). It serves as a data feeder, reading data from a high throughput stream or memory map and then supplying structured data to subsequent subsystem modules. It may also be used to directly feed generated data into the signal chain, functioning as either a data spoofer or as a playback mechanism for archived data. Although the modules in this configuration act as

data entry points for their signal chains, a direct use of the input data channel is disabled. This is due to the unstructured input data typically requiring a custom implementation by the module's developer. Control channels allow for custom details of source data to be specified, e.g. polling intervals and parameters for throughput throttling.

#### 5.1.3.2 Processor configuration

This configuration acts as a throughput block by converting incoming data to the desired output format. Herein, it functions analogously to the classic Transforming Element block (see section 2.2.2.3). When MQTT's topic filtering is insufficient or unavailable, a Processor configuration may be used to create a more intricate Selector block (see section 2.2.2.2). These configurations will typically serve as the core components of signal chains and host the most computationally demanding tasks, such as high-level signal processing, tracking, and classification. Control channels allow for configured parameters relating to aspects of data processing to be specified, e.g. signal processing thresholds.

#### 5.1.3.3 Consumer configuration

This configuration is analogous to non-HMI implementations of the classic Consumer block (see section 2.2.2.4). It acts as a data sink, for example, transferring archived data to the subsystem node's local storage. Although the modules in this configuration act as data termination points for their signal chains, a direct use of the output data channel is disabled. This is due to output data not being intended for subsequent consumption by a conformant signal chain module and typically requiring a custom implementation by the module's developer. Control channels allow for custom details of data output to be specified, e.g. the storage path of archive data destination for external streaming.

#### 5.1.3.4 Control configuration

This configuration has no direct data input or output, with exclusive I/O through the control channels. Although it is not strictly analogous to any classic Signal Chain block, it is necessary for interop control of lower-level systems. Control channels allow the manipulation and monitoring of associated systems and processes, e.g. antenna rotator.

### 5.1.4 Subsystem modules

Subsystems can be defined dynamically, with the broker serving as a means for modules to describe their control schema to other parts of the system. Such modules may be geographically distributed, but all connect to a central broker for wiring into a wider signal chain.

Subsystems should allow for management over their configurations and processes through the broker and data transfer via specified input and output channels. To facilitate this, subsystem modules can be considered as variants of a basic templated design. Subsystem modules may be written in any programming language that supports MQTT and TCP/IP (see section 4.2), with notable examples currently including C [78], C++ [79], Python [80], Julia [81] and Matlab [82].

A Python assembly, which handles all the core subsystem tasks, is provided to simplify subsystem development. Memory maps ensures wider language compatibility and interoperability, allowing developers to work in a familiar environment whilst enabling certain types of processing for which Python may not be well suited. The assembly ensures adherence to the interface control schema, providing abstracted messaging to the subsystem developer.

Helper functions provide core functionalities not inherent to Python (e.g. generalised timestamps, colour coded console output, lookup tables, and enums), as well as base classes for implementation uniformity (e.g. common endpoint definitions and events).

If a subsystem connects to an active signal chain, it subscribes to the appropriate topics as directed. Subsystems should independently manage their underlying components and perform any background workflows that are required.

Figure 5.5 depicts an activity diagram showing subsystem startup from a co-located configuration file. If the subsystem determines itself a component of the active signal chain, it looks for earlier subsystems and use their designated output as input. The subsystem also publishes the full schema of its output data and associated controls.

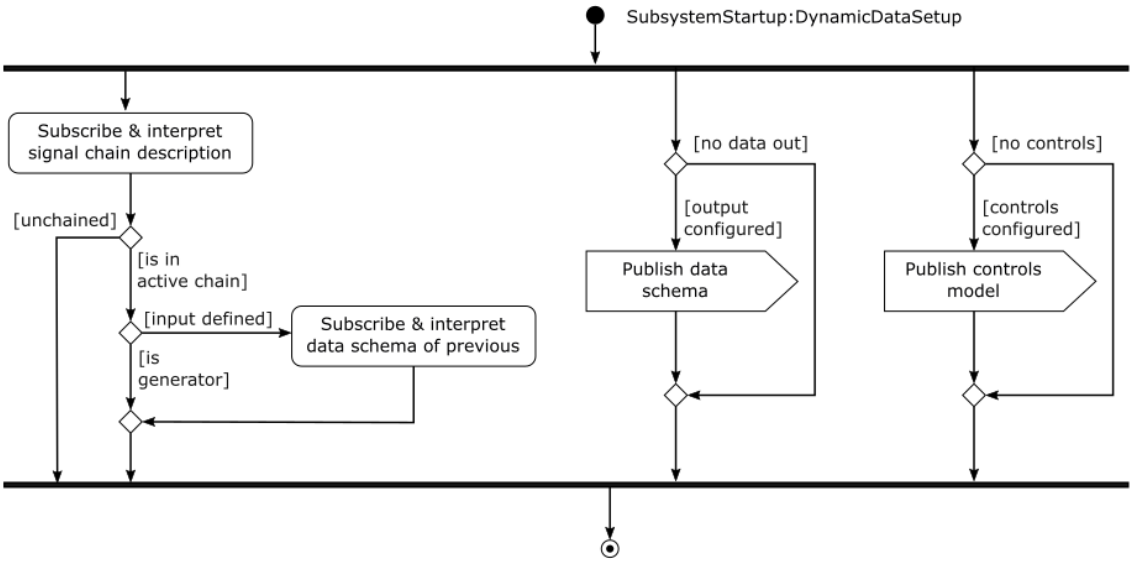


Fig. 5.5. Activity diagram for subsystem setup from the associated configuration file

Subsystems broadcast their presence, output data links, and current status periodically from startup, permitting their configuration to an active signal chain.

Figure 5.6 depicts the basic components and connectivity required for a subsystem to be fully integrated into the overall data model describing its signal chain. The Controller (see section 5.1.4.1), Input Channel (see section 5.1.4.2), Process (see section 5.1.4.3) and Output Channel (see section 5.1.4.4) components are described in subsequent sections.

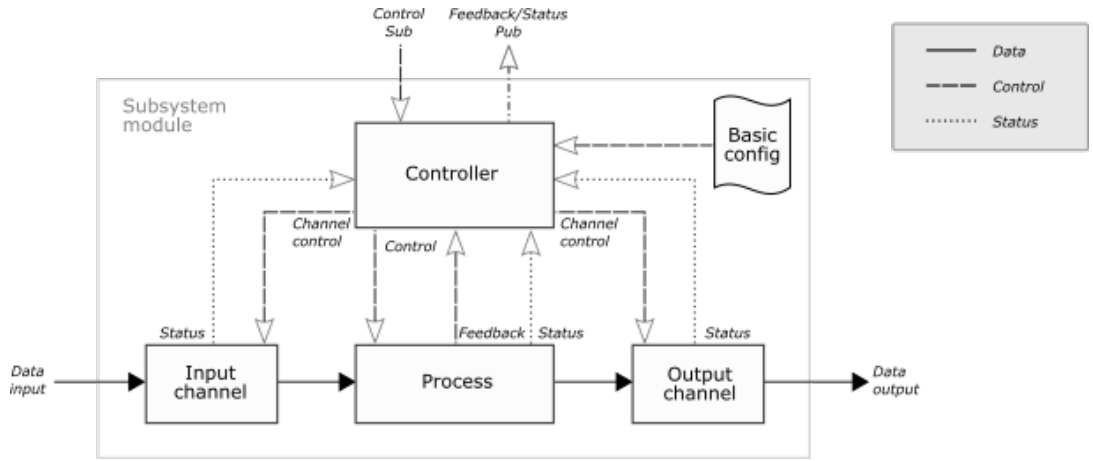


Fig. 5.6. Component diagram of a fully functional generic signal chain subsystem module

#### 5.1.4.1 Controller component

This component is in charge of publishing the subsystem's control schema, as acquired from a YAML (YAML Ain't Markup Language) [83] configuration file, which instructs how external entities may manipulate the subsystem's operations with a data and control schema definition. The subsystem also uses the controller to interact with its core internal components for monitoring and automated management.

#### 5.1.4.2 Input Channel component

This component is in charge of providing a data input point and also unpacking data for further processing. It may be configured to accept no input, subscribe to data on an MQTT broker, or receive data straight from a TCP/IP stream.

#### 5.1.4.3 Process component

This component contains the subsystem's core logic and serves as its primary data manipulator (e.g. tracking logic for a Tracker). It is typically the only component that requires unique code, with other components managed by the assembly (as instructed by the subsystem's basic configuration at startup). As such, it should commonly be the sole focus of subsystem development.

The Process component monitors changes to its configured controls' data model and make adjustments as accordingly. It also determines and describes its state to the Controller component, all while preserving as much stability as feasible by throttling and overriding parameters as warranted (e.g. disabling functionality for performance or safety).

#### 5.1.4.4 Output Channel component

This component is in charge of packing data after processing and, when configured as an MQTT publisher, may be expected to deliver multiple output data types simultaneously. It may be configured to output nothing, publish data to a particular MQTT topic, or stream data directly as a TCP/IP server.

## 5.2 UI design and layout

This section provides a visual design for the HMI of the radar system detailed in section 5.1. In specific, it considers the broad functional outline provided in section 5.1.1 for the browser-based radar HMI, as well as the common HCI principles in section 2.6.

To be fit for purpose, the design should allow for the dynamic creation of sequential signal chains from actively broadcasting subsystems. Multiple signal chains should be available for selection and configuration, along with some control over the rendering of associated map elements for attention management. Each subsystem manages the specific controls required for their, along with the styling of associated map elements. Such control over a display would typically be under the exclusive domain of an HMI designer but is delegated here due to the nature of the underlying implementation and a focused data-driven approach.

### 5.2.1 Base design

Figure 5.7 presents a simplified sketch of the intended browser-based HMI design (i.e. with all controls collapsed, providing an uncluttered view of map-based data).

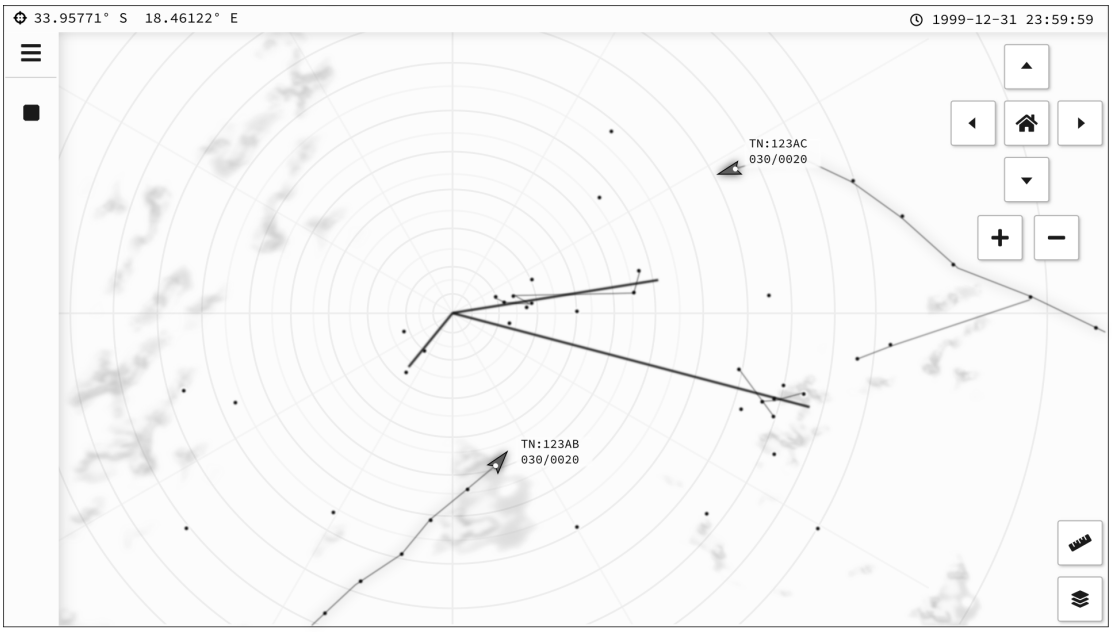


Fig. 5.7. Design for browser-based HMI in its decluttered state

The top bar in figure 5.7 shows the cursor location, as well as the current date and time. Controls in the top-right corner enable panning, zooming, and returning to a home location. The bottom-right shows other map controls, such as layer toggles and a measuring tool. Signal chain controls are on the left and may be extended for specific subsystem control and setup (detailed in subsequent sections). The current state of the active signal chain and control thereover is provided in all views, allowing for quick action where stability or safety becomes impacted.

Prominent map elements include heat maps, plots, strobe lines, and tracks (composed of a trail, selectable symbol head, and label). Track labels should default to the simplified NATO guidelines for air symbol labels, with an identifier on the first row and the target bearing/speed on the second [84] – but with SI units for overall consistency.

Given the great variety of devices that allow for the use of a browser-based HMI, the design should be adaptive to various device resolutions and aspect ratios. With consideration to touch input with mobile device use, specifically, care should be taken to appropriately size and space interactive elements (see section 2.6.1.2).

Figure 5.8 shows how the design adapts to mobile devices. With exclusive touch input assumed for these small-screened devices, some camera controls may be omitted. The current date and time may also be removed in portrait mode, with auto-rotation used to manage access to certain features and more detailed views.

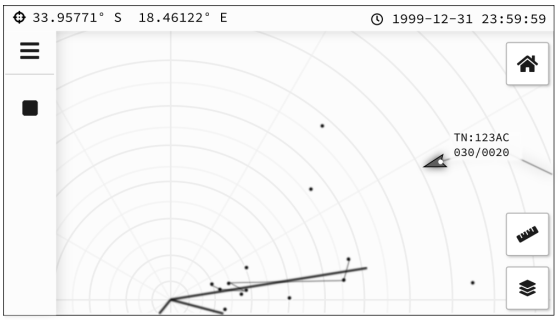


Fig. 5.8. Design presented in figure 5.7 as adapted for mobile device use

### 5.2.2 Subsystem controls

The menu panel in figure 5.7 can be extended to show subsystem cards with their specific status indicators and controls. These may be further expanded to reach fine-grained subsystem-specific controls, as shown in figure 5.9, along with broader HMI display options such as a light/dark mode switch at the top.

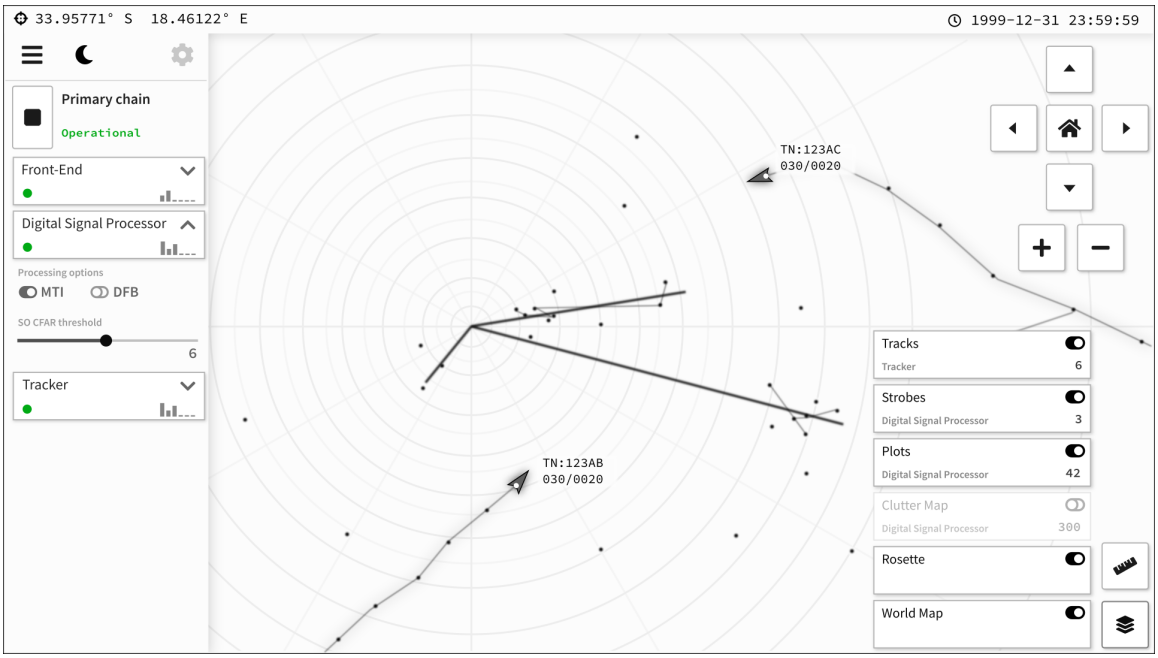


Fig. 5.9. Design of HMI with expanded subsystem controls

Each subsystem defines its control views, allowing for more precise manipulation of its operations. Toggles, radio buttons, sliders, and text boxes are dynamically added to the subsystem panels as defined in their configuration files (see Appendix B). Control collections should follow the Hick-Hyman law (see eq. 2.2), though here is the responsibility of the subsystem developer that provides this layout.

The active map layers control, shown expanded in the bottom-right of figure 5.9, provides access to a toggleable collection of visual layers available for the active signal chain. It also presents a counter that indicates the number of items in the current scan for the associated layer.

### 5.2.3 Signal chain setup

Figure 5.10 shows the HMI in edit mode, which is only accessible when the active signal chain is halted. All signal chains may be configured in this mode, including their origin and range, related subsystems, and the opacity of accessible map layers. Chained subsystems may be removed from the active signal chain, and any unchained broadcasting subsystem (shown in light red) added.

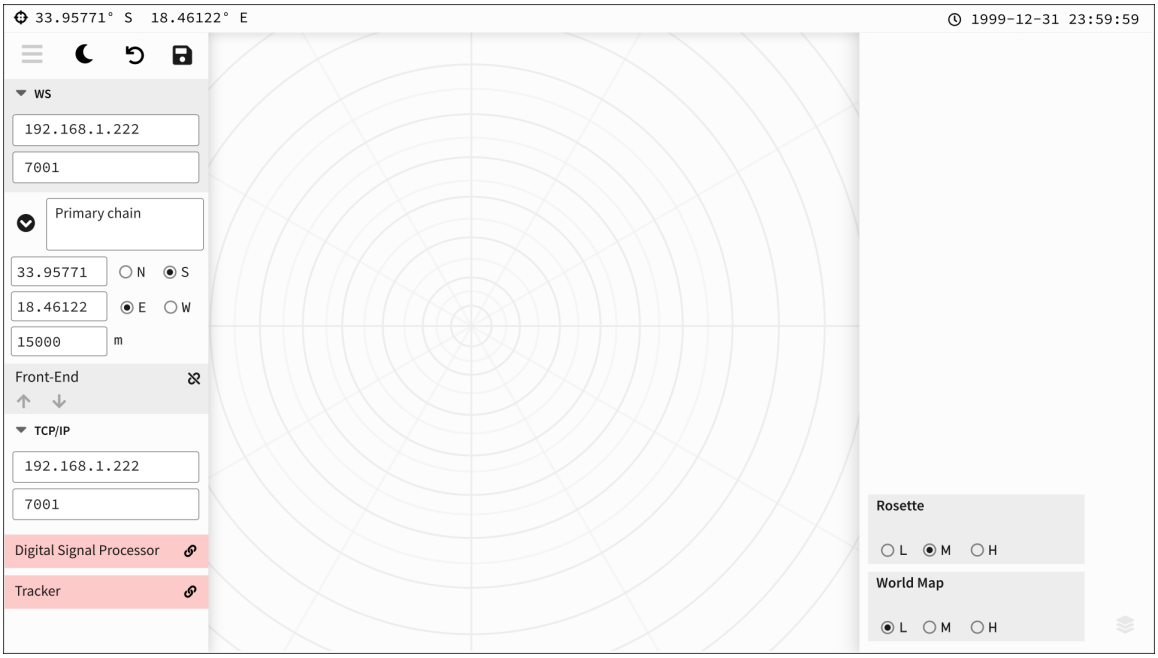


Fig. 5.10. Design for signal chain setup from broadcasting subsystems

As shown in figure 5.11, once subsystems are chained in, they may be ordered and configured as appropriate. It also allows setting the transport protocol between subsystems to TCP/IP or an appropriately configured MQTT endpoint to enable non-WebSockets traffic where HMI rendering is not required (as indicated in section 4.2).

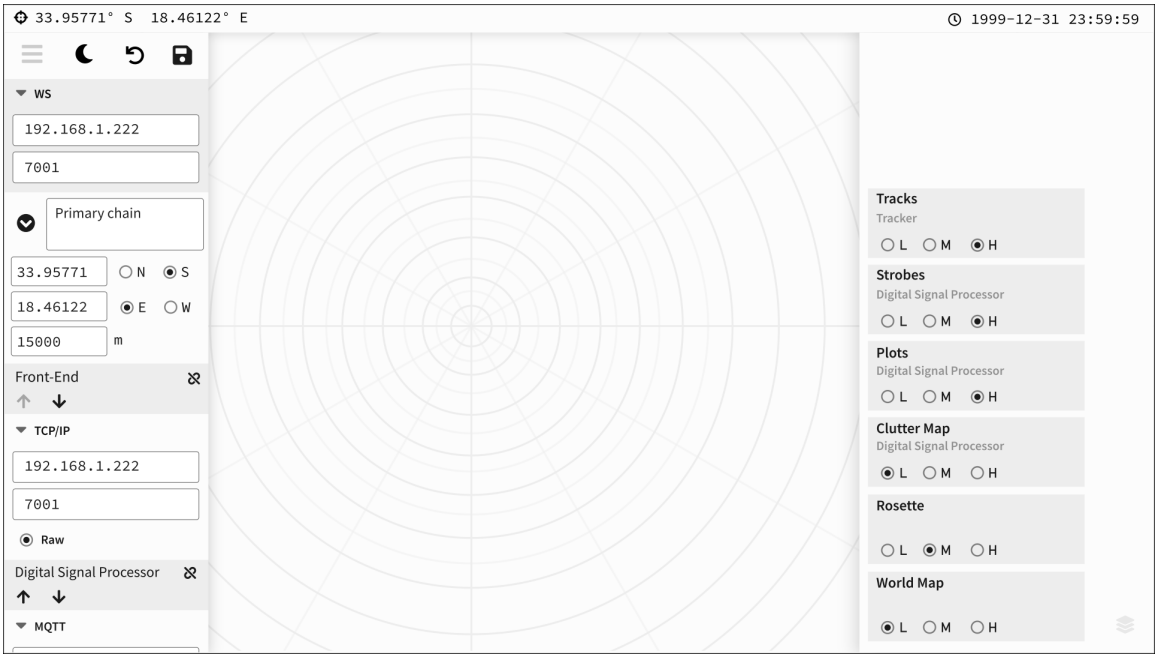


Fig. 5.11. Design for HMI in signal chain edit mode

Figure 5.12 shows a list of configured signal chains after selecting the drop-down next to the name of the current chain. Here, chains may be created, a configured chain selected as the active chain, or an existing chain removed from the system.

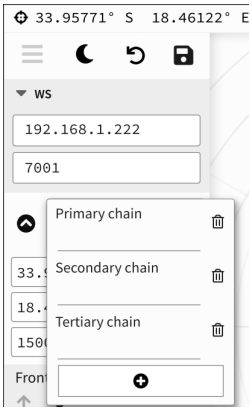


Fig. 5.12. Control design allowing signal chain selection, creation and removal

# Chapter 6

# Results

This chapter explores the general usability and versatility of client and subsystem environments using a selection of demonstrative signal chains. A series of tests, with a subsystem directly spoofing common high-level data elements (e.g. plots, strobes, and tracks), establishes their overall performance characteristics (i.e. the effective throughput and evidenced bottlenecks of the disparate components). A summary of results is provided in section 6.3, with conclusions and implications subsequently discussed in chapter 7.

## 6.1 Usability outcomes

The outcome of the designs presented in sections 5.1 and 5.2 show the evaluations intended from chapter 3, with some consideration to section 2.4. It shows various task-oriented systems using signal chains consisting of basic subsystem blocks. The HMI also shows reasonable hardware and software cross-compatibility within various environments.

### 6.1.1 Client usage versatility

This section emphasises the general ability to use the client on a limited sample of devices. The HMI was verified using the Edge and Chrome browsers on desktop (Windows 10) and mobile (Android) devices, using mouse or touch input as appropriate.

#### 6.1.1.1 Desktop device use

Figures 6.2 and 6.6 show the realised HMI closely following the design in figure 5.7. It allows the signal chain setup shown in figure 5.10 for the broad tasks in section 6.1.2.

PWA adoption is still in its early stages, therefore browser and operating system developers' priorities continue to shape it. As of this writing, full usage is generally available, with notable exceptions being Mozilla Firefox [85].

#### 6.1.1.2 Mobile device use

Unless caused by variations in processing performance, disparities between desktop and mobile devices are often readily remedied. Since style rules may be interpreted differently across devices, extensive testing is still needed to ensure UX consistency.

SVG use is limited due to frequent style issues on mobile devices and the HTML canvas outperforming it. As of this writing, full usage of PWA is generally available, with notable exceptions being non-Safari browsers on iOS [86].

Figure 6.1 compares the light and dark modes on a representative mobile device (with 731x411 pixel screen) in landscape mode, as emulated with DevTools on a desktop-based Chromium browser. Note the resolution-dependent camera control in the top-right corner, showing the application adapting to the available screen area as from figure 5.8.

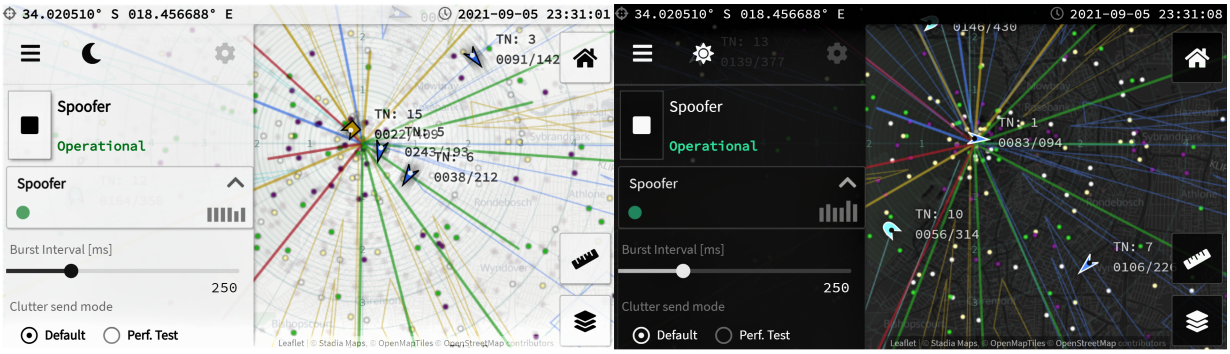


Fig. 6.1. Comparison of light and dark modes for landscaped mobile displays

Visual elements with high processing needs have lower overall responsiveness on mobile devices than desktops, reflecting the computational contrasts of these device types.

Using components such as Leaflet's heatmaps (as with plots and clutter) may also result in undesired behaviour. These require regular updates of relatively large data blocks to the HTML canvas on the browser's presentation thread, resulting in polled update controls (e.g. the current timestamp and general status indicators) flickering noticeably.

Wider polling intervals make such flickering less apparent and reduce the overall processing load, which is especially beneficial on lower-performance mobile devices. Hence, the exclusive use of persisted elements, such as tracks and strobes, is advised where mobile device use is a particular focus. As such, subsequent testing emphasises desktop use.

### 6.1.2 Demonstrative signal chains

The implementation shows a high degree of versatility with regards to the type of high-level radar systems that may be realized. This is achieved through a single-flow signal chain created from the component configurations defined in section 5.1.3.

Some of the systems that may be created with representative signal chains are demonstrated here to show the versatility of the solution. These signal chains all make use of a Python-based assembly that eases the realization of section 5.1.4's design. Chains demonstrate commonly expected radar functionalities such as the ability to view/record data, the playback of recorded data, and multi-stage processing flows.

Appendix D lists and details distinguishing logic of Process components (see section 5.1.4.3) for the subsystems discussed in this section. Specific component logic is for demonstrative use and is limited to subsystems that do not require extensive interprocess communication for internal processing (e.g. as with the alternate Matlab-based tracker subsystem).

The Controller (see section 5.1.4.1), Input Channel (see section 5.1.4.2) and Output Channel (see section 5.1.4.4) components are implemented within a base Python assembly, and is instructed through a subsystem specific configuration file (see Appendix B). It ensures that subsystems correctly implement their respective data models to the broker (see Appendix C), allowing their dynamic use in various signal chains.

The Cape Town campus of UCT, in the Western Cape province of South Africa, is used for geo-referenced examples due to familiarity and having adequate ADS-B and AIS traffic.

#### 6.1.2.1 Viewer

The solution may be used to build a simple data viewer that provides quick insight into specific systems. In general, this may be achieved with a subsystem-based data translator, which converts data into a format that the framework recognises. The subsystem may then be used in a single element signal chain to give feedback and control over the interpreter.

Appendix D.1 shows the specific processing logic used to implement a polled request for ADS-B tracks to the OpenSky Network's [62] REST API, which allows batched queries at set intervals. After unpacking the response, a message is delivered to the HMI for each retrieved track through the specified output queue. See Appendix B.1 for the associated subsystem configuration file and Appendix C.1 for the auto-generated data model.

Figure 6.2 shows the output for tracks in a specified bounding box. The HMI presents in its light mode on a desktop monitor (1920x1080 pixels), with tracks as defined in Appendix 2.6.1 and contextual popups displaying ADS-B specific details as interpreted.

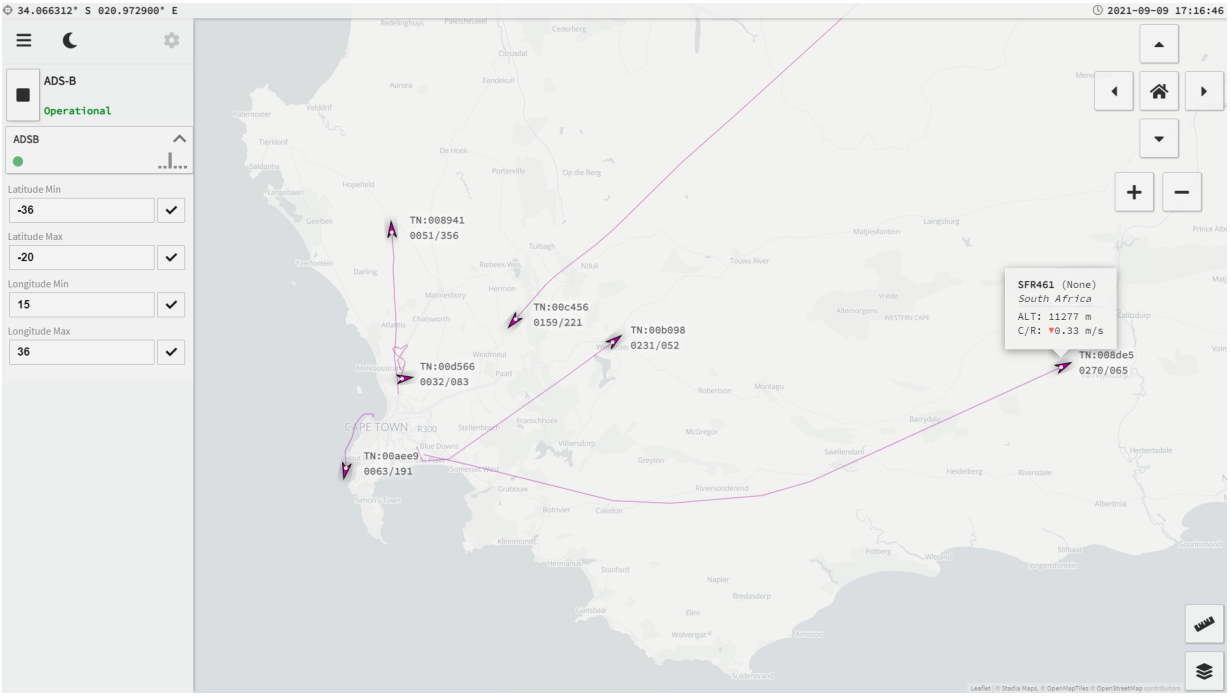


Fig. 6.2. Screenshot of real-time ADS-B data from the OpenSky Network [62]

The implementation works well for the regional display of ADS-B tracks, with processing spikes as expected due to the batched nature of OpenSky Network queries.

#### 6.1.2.2 Dedicated control

Control changes may be written to a memory map for interprocess communication. Where the external process allows read-only memory views, it may safely query the memory map for changes at regular intervals. A Matlab session is used as the external party here, which is sufficient to show operation due to the common nature of such communication.

The specific processing logic listed in Appendix D.2 shows the use of a matched memory map file, which remains open for exclusive write access. Two control update mechanisms are implemented: a poll-based approach that regularly updates the memory map and Observers for event-based memory map updates. See Appendix B.2 for the associated subsystem configuration file and Appendix C.2 for the auto-generated data model.

Figure 6.3 depicts the panel for such a Control subsystem, with the matched mapping in memory depicted to the right.

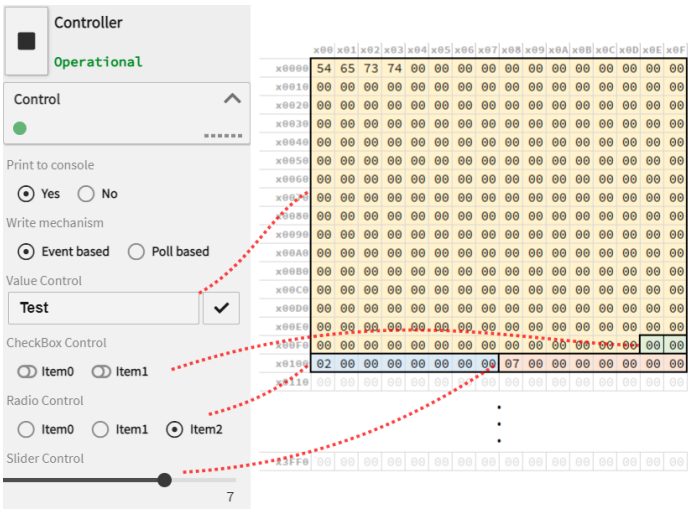


Fig. 6.3. Screenshot of subsystem panel, and indication of values reflected to a memory map

Control updates through both event-based and polled-based mechanisms are determined viable for extra process communication, with the choice up to subsystem developer needs.

#### 6.1.2.3 Recording

This signal chain consists of an Emulator subsystem that sends spoofed "raw" data via

TCP/IP to a Recorder subsystem. The Recorder's primary responsibility is to store received data in memory map archives for subsequent playback. Each archive contains a header with metadata to ensure correct interpretation by playback mechanisms (e.g. message size, rotation period, sensor origin, file sequence etc.).

The specific processing logic listed in Appendix D.3 shows the emulation of detections of a defined intensity and speed at a position relative to a defined origin. Emulated detections are provided for targets following an outward spiral alongside a controlled number of spurious detections depicting clutter. Data transfer occurs in blocks with a preset azimuth span and interval, with the number of emulated detections per block controlled from the HMI. See Appendices B.4 and B.6 for the associated subsystem configuration file and Appendix C.3 for the auto-generated data model.

Although there is no data with direct map representations here, throughput may be monitored using the bar chart on the subsystem panels, as shown in figure 6.4.

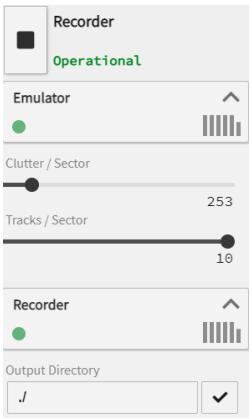


Fig. 6.4. Screenshot of subsystem panels for the archival of emulated data

Data may rapidly collect, resulting in large files that are difficult to use and prone to corruption; therefore, archive files are restricted to about 131 072 entries (pending a script update). When this entry limit is reached, a new archive is created, with new entries then being added thereto until the size limit is again reached. Furthermore, completed archives may be moved to long term storage without interfering with ongoing recording activities.

As such, and since continuous recording may quickly consume a lot of space on active systems, an automated archival process for finalized recordings should be considered where long-term data storage is required.

#### 6.1.2.4 Playback

The archives from section 6.1.2.3 may subsequently be read by replay mechanisms, with a subsystem that is implemented as a classic Signal Chain Generator (see section 5.1.3.1).

The "raw" data points offer a sensor-relative range and azimuth, with latitude and longitude ultimately required for presentation. Although the HMI can perform the conversion, bulk conversion by a subsystem is preferred. The Processor subsystem's main logic uses a threshold to filter out low-level returns and is Matlab-based, which is more performant than a wholly Python-based subsystem here. See Appendices B.3 and B.5 for the associated subsystem configuration file and Appendix C.4 for the auto-generated data model.

Figure 6.5 depicts a signal chain with a Data Feeder subsystem reading "raw" recorded data into a subsequent processor. The intensity of clutter points derived therefrom is shown using the Magma palette. Plots are similarly colour-coded by the intensity of their return, with green indicating a definite return, yellow an intermediate, and magenta a weak return.

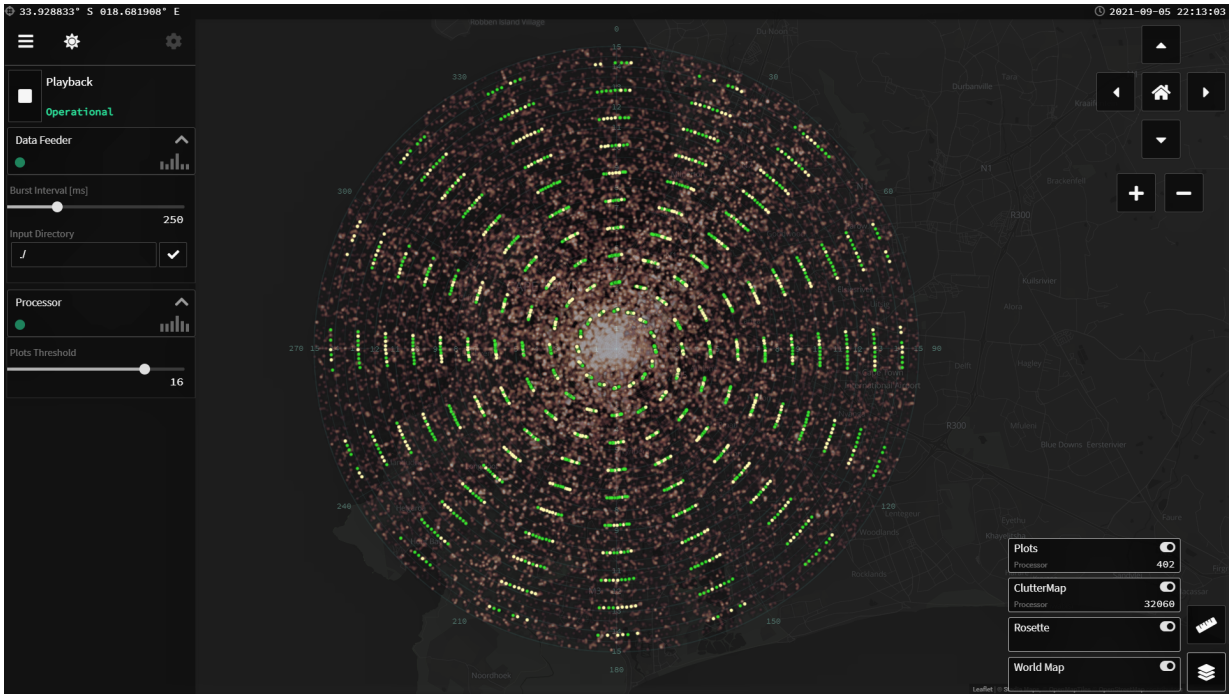


Fig. 6.5. Screenshot of plot and clutter input from archived data

When recording lower-level data, a large number of files may quickly accrue. For lengthy sessions, it may be prudent to compress the archives with direct playback therefrom.

#### 6.1.2.5 Full chain

This section shows a basic implementation of the signal chain presented in figure 5.1. The Emulator and Processor subsystems feed into a terminating Tracker component here. Plot messages retain the range, azimuth, velocity, and detection time of the initial detection, since these may be required for subsequent processing.

Plot speeds are used to create a detection window in which to look for associated plots within following scans. A preliminary track may then be created by roughly matching the speed of plots near the expected detection time of subsequent target detection. When enough detections are amassed on these preliminary tracks, they are accepted as legitimate tracks and submitted to the HMI.

In this way, the track length is effectively used as a substitute for the initial quality. The track quality is limited to allow for a more gradual removal of high-quality tracks by prolonging them until they fall below the defined quality threshold.

The Tracker subsystem's main logic is Matlab-based due to the need to efficiently determine patterns from numerous plots and ensure that subsystem processing does not become a primary bottleneck. The Emulator subsystem is as described in section 6.1.2.3, with the same Matlab-based Processor subsystem described in section 6.1.2.4 employed. See Appendices B.4, B.5 and B.8 for the associated subsystem configuration files and Appendix C.5 for the auto-generated data model of this signal chain.

Development is greatly facilitated by the fact that necessary modifications to the base templates, as described in Appendix D, typically only affect the execution loop. Simple modifications of the base templates are sufficient for initial prototyping, but performance can be increased by progressively adding more complexity. In specific the logic for the subsystems used here provides expanded functional logic to allow inter-process communication, but even so still closely follow the base code templates.

Figure 6.6 conveys track classifications through both colour and shape, with rounded light blue symbols representing low-velocity tracks and sharply angled dark blue symbols representing high-velocity tracks. The detection window for a track is detailed in its contextual popup. The tracker panel controls enable dynamically specifying the desired number of valid tracks for which to generate detections (not shown here).

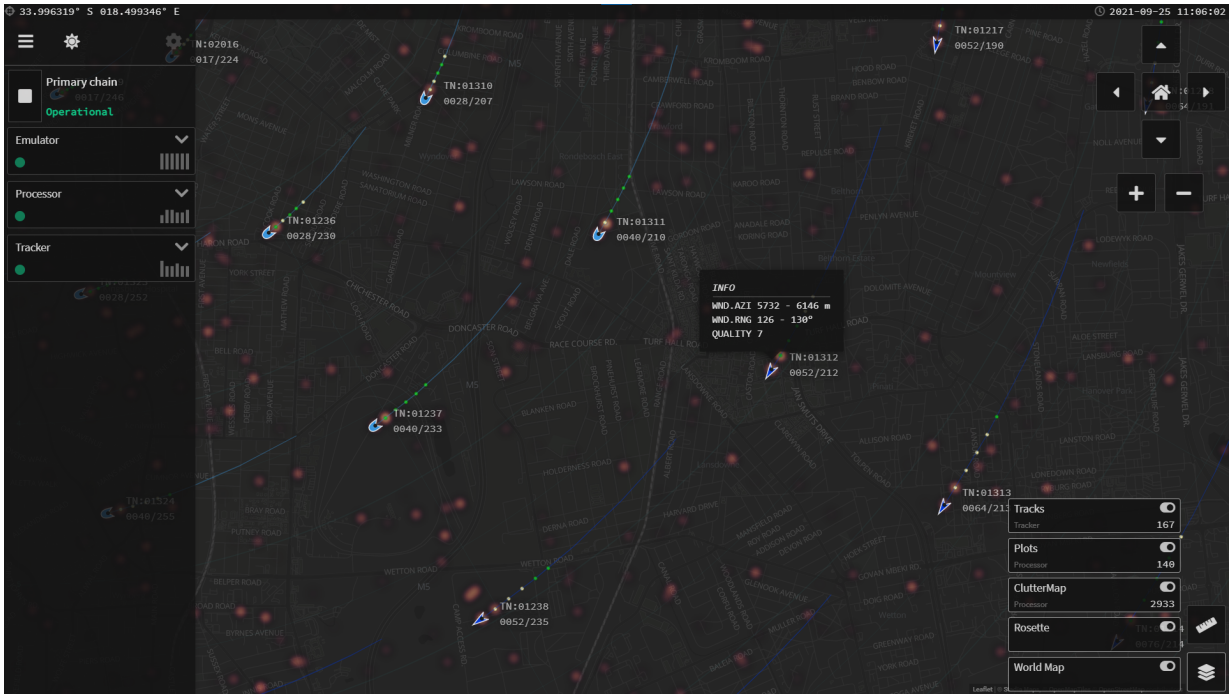


Fig. 6.6. Screenshot of the actualized signal chain from figure 5.1

Any language that supports the necessary protocols may be used to create subsystems, although the supplied Python assembly is the most convenient. Where cross-language support is needed, Python may serve as a supervisor, using operating system commands to launch processes and memory maps for interprocess communication.

Matlab is a suitable supporting language as it offers direct type compatibility and active share engine control to Python. As seen in section 6.1.2.5, this allows for higher-performance data processing to be passed onto Matlab, with Python used to provide general subsystem management and a messaging interface.

#### 6.1.2.6 Making use of a cloud environment

Microsoft's Azure cloud services are used here to demonstrate the use of IaaS/PaaS/DaaS. For remote testing, the Azure App Service (South Africa North) is used as the PaaS-based web server. The core broker and its related subsystems are hosted on an Ubuntu 16.04 Virtual Machine (VM) as part of Azure's Standard B1s (US East Coast) IaaS offering.

The specific processing logic listed in Appendix D.4 shows the directly handled interpretation of emulated data to clutter and plot points. It requires determining a detection's coordinate from its sensor relative position and filtering these by an HMI controlled plot threshold. The emulator is the same as section 6.1.2.3 but co-deployed to the Processor subsystem on the cloud. See Appendices B.4 and B.5 for the associated configuration file and Appendix C.5 (absent tracker) for the auto-generated data model.

Figure 6.7 demonstrates a realized cloud deployment with a signal chain that consists of an Emulator feeding into a Python-based Processor. Clutter and plot points have the same intensity defined colour-coding as in section 6.1.2.4 but with the HMI in light mode here.

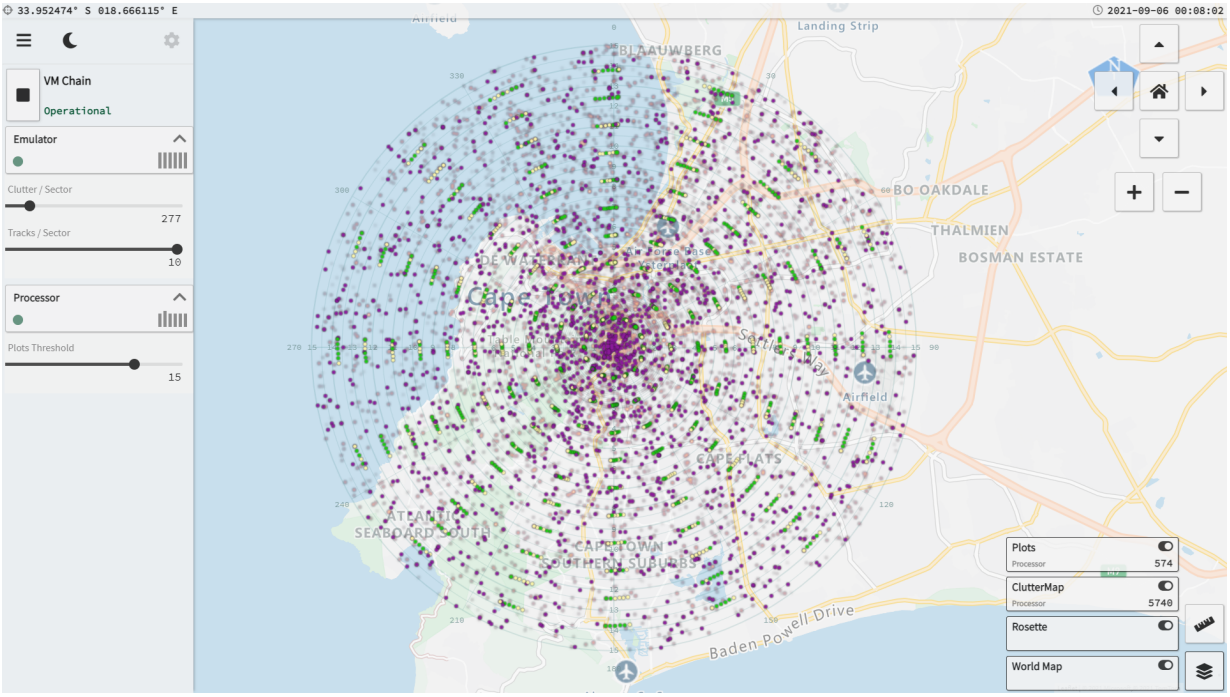


Fig. 6.7. Simple signal chain demonstrating the use of a cloud-based deployment

Cloud-based services have strict security demands, necessitating encrypted communication for interaction with its secured environs. As such, the secure equivalents of all protocols are used (i.e. HTTPS for the site, WSS for push data, and MQTTS for the broker).

PaaS use often requires configuration files and specific directory structuring. In addition, some services, such as the WMS (see figure 6.7), may be limited to the provider (e.g. Azure maps). These are often priced separately and of limited throughput. It also necessitates decision logic due to differences between on-premises and cloud-based deployments.

## 6.2 Performance metrics

This section looks at reasonable performance expectations for the system, with some consideration to section 4.2. An on-premises web server and MQTT broker combination (as in sections 6.1.2.1-6.1.2.5) is compared to a cloud-based deployment (as in section 6.1.2.6) to explore how these deployment options impact overall performance.

The primary focus is to determine the constraints within which the framework effectively functions with the use of mainline hardware and host environments.

The client and local environments are as specified in chapter 4, with the remote environment as defined in section 6.1.2.6. While not explicitly considered here, mobile devices may simply be regarded as less capable computing devices, with comparable impacts to responsiveness and effective throughput (at least where elements can be used without issue, see section 6.1.1).

The 1 Mb/s connection to the cloud-based VM is a throughput limitation that may be addressed by acquiring extra network capacity, but a basic account configuration is assumed here. The significantly increased latency is highly dependent on the cloud provider's infrastructure and problematic to address with a physically distant VM host.

### 6.2.1 Startup-time

General system initialization times are considered here, with a particular focus on HMI startup. The web server process may be configured to start concurrently with the operating system on which it is deployed. When compared to a host's startup time, the web server initialization can effectively be considered as near-instantaneous.

Figure 6.8 shows a comparison between loading the web application from an on-premises web server, against an Azure-hosted web server. The first scenario includes disabling local resource caching and removing previously loaded application resources, showing full initial retrieval or usage in the absence of PWA capabilities. Subsequent scenarios consider HMI startup when PWA based caching is enabled, with and without a connection to the originating web server.

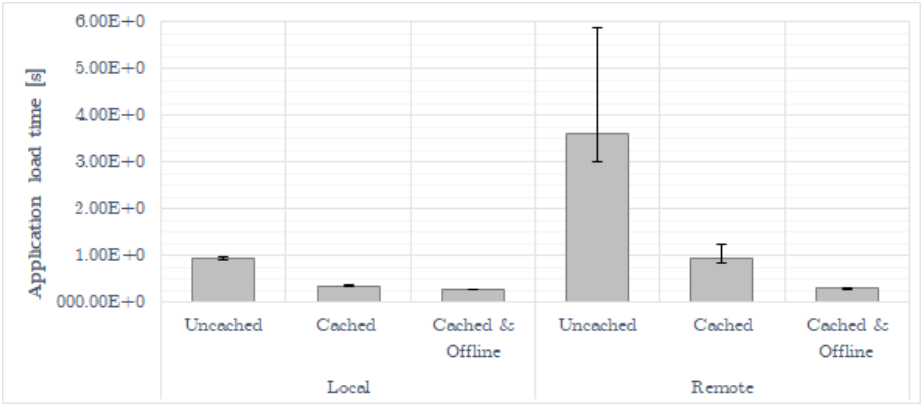


Fig. 6.8. Comparison of browser-based HMI startup times with and without PWA use

From figure 6.8 it is clear that HMI startup is highly dependent on throughput to the web server, but that subsequent access with both local and remote deployments can be sped up significantly with caching enabled.

The longer cached startup times with an active web server connection may be ascribed to the DNS lookup and comparison of locally cached resources to those on the web server. The difference between local and remote use is negligible where application resources have been cached and the web server connection is unavailable and may be attributed to different WMSs being used to retrieve map tiles.

Once the application has launched, the system configuration, which includes the current chain, may be accessed. An additional 1 second to the overall HMI startup time may be assumed with an on-premises broker and 2-4 seconds with a remotely hosted broker (highly dependent on the connection).

### 6.2.2 Usage performance

To be deemed functional, the client scene must be kept in sync with the incoming data, protecting the system against a backlog of unhandled changes and keeping the view current. The protocol constraints detailed in chapter 4 should be considered, but other factors may well prove more restrictive. Three significant bottlenecks apply to the browser-based application in particular: incoming message rates, payload processing, and canvas updates.

Data rates should not be so high that CPU processes are over-allocated, as this may cause stuttering and subjectively poor HMI responsiveness. Actively changing DOM elements are skipped until the next render/layout cycle, resulting in visual artefacts when panning or zooming, such as the zoom level on all layers not being in sync. Excessive canvas update rates may even result in the HMI becoming completely unresponsive.

This may be prevented to a significant extent by ensuring that render updates are delivered in bulk at intervals greater than 100 ms. To that purpose, data is stored in application dictionaries and the canvas is updated from them at polled intervals.

As such, for a 4-second rotation period radar, data transmission intervals of 250 ms (i.e. 22.5° blocks) may be considered. The signal chain consists of a single Python-based Spoofer subsystem that allows for customization of the transfer profile and selective injection of clutter points, plots, strobes and tracks. See Appendix B.7 for the subsystem configuration file and Appendix C.6 for the auto-generated data model. These allow for the performance constraints and load transfer characteristics of select elements to be determined here.

#### 6.2.2.1 Plots and Cluttermaps

These elements are non-interactive and organised into layers, with each layer representing a fixed slice of time equal to a full scan period as defined in the subsystem configuration.

Such grouping improves performance by allowing styling to be pre-coupled to a layer, with point data then manipulated to generate particular views. It also ensures that only the current layers are being updated, reducing DOM interaction to smaller structures. This layer structure also allows for the mass disposal of older data after a certain number of rotations, reducing excessive buildup.

Both plots and clutter make use of Leaflet's heatmap plugin, allowing for a similar approach, albeit their layers are handled and styled slightly differently. Since previous clutter layers may obscure subsequent layers, newer layers are kept on top; this render order switching, together with the difference in the plot (50 B) and clutter (21 B) record sizes, accounts for performance variations here.

With a local deployment, about 536 000 plots/rotation (6.70 MB/s, 0.9% error) and 768 000 clutter points/rotation (4.03 MB/s, 0.9% error) may be reached before HMI use

and performance suffer significantly - roughly serving to estimate the allowed worst case. The cloud-based deployment reached 2 512 plots/rotation (125 kB/s) and 20 976 clutter points/rotation (110 kB/s), due to IaaS throughput constraints.

Due to slight timing differences, the active count in each rotation may not entirely line up with the current layer when calculated. At heavy loads, active counts can vary by up to about 20 000 points/rotation (about 4%) at these rates but still roughly average at the given levels where stabilized over a period of at least 1 minute.

The upper limits of the local deployment are lower than expected for the protocol and mostly determined by HMI responsiveness, indicating that the bottleneck is on the client-side and device-dependent. Plot throughput does approach the protocol limit (see chapter 4) showing near optimal use, but should be interpreted with consideration to the rendering performance of test hardware (see section 4.1).

#### 6.2.2.2 Strobes

These non-interactive elements are handled individually, with a unique strobe key enabling post-creation adjustments to placement and colour. Each strobe record is about 63 B in size.

From in-browser CPU allocation, creating new strobes is seen to be costlier than updating existing strobes. A sudden influx of data may cause a blockage, resulting in a loop in which strobes time out and the scene never synchronizes with input again.

The scene consistently remains in sync as long as the strobes are created in time with the incoming data; this rate has been determined to 5 120 strobes/rotation (80.64 kB/s, 0.2% error) for both local and cloud-based deployments.

The system may still recover with an excessive influx, as queued updates are processed faster than the update rate once the strobes are created. A higher rate is then sustained, with a local deployment reaching a clean start rate of 11 600 strobes/rotation (182.7 kB/s, 0.9% error).

When the strobe count is continually raised without triggering a cascade, significantly higher rates, up to 36 800 strobes/rotation (579.6 kB/s, 0.4% error), may be obtained.

#### 6.2.2.3 Tracks

These interactive elements consist of a combination of elements, requiring several DOM nodes that are keyed to allow post-creation updates. To ensure that dead tracks are pruned, the latest update time of each track is periodically compared to a keep-alive time.

With the track head being interactive, the cost of utilising these elements is relatively high. The subsystem developer may also fully customise the contents of the associated track popup, necessitating a lengthy string in each record. As a result, track records are relatively big (212 B), which affects the throughput efficiency since a filled 16 kiB payload sees about 60 B of unused space.

The application processing performance limits are examined with track rendering disabled. The number of track trail updates where application use becomes degraded is then tallied. To more definitively assess where system performance becomes significantly impacted, the in-browser CPU is monitored for continuous spikes suggesting excessive allocation, as likely to interfere with responsiveness and general application use.

Figure 6.9 shows two common cases, the first for a 4-second scanning radar (dots) in which track updates occur in angular blocks throughout the rotation. The second case is a 1-second tracking radar (crosses), which sees all active tracks updated simultaneously. The inverse power-curve approximations for the scanning (solid) and tracking (stipped) cases are indicated along with vertical lines for their upper usage rates with rendering enabled.

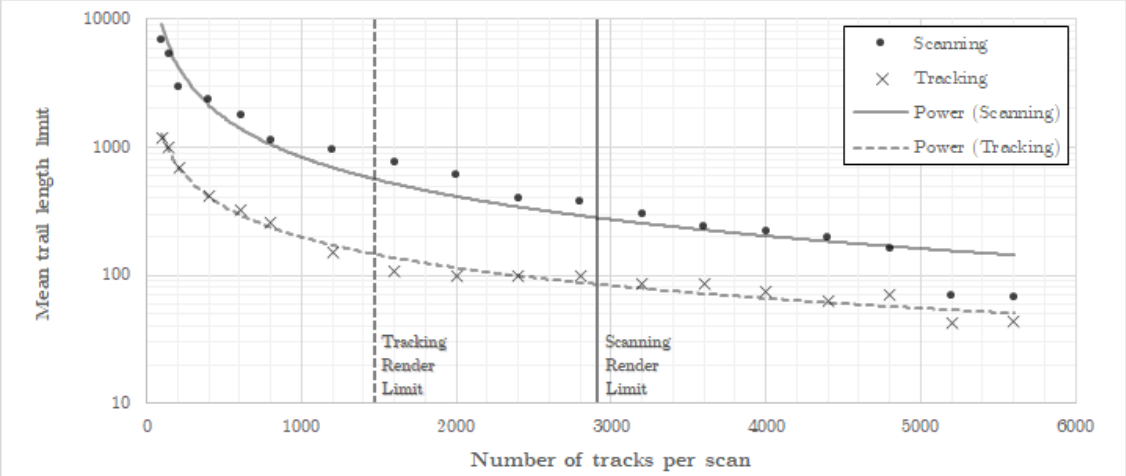


Fig. 6.9. Stable track length and entrant render rate limits for considered use cases

Trails extend in length over time, limiting the number of tracks that can be kept active due to increased memory usage and update costs, as seen in figure 6.9. This is often the most essential factor to consider when evaluating extended duration tracks here.

Although the addition of HTML-styled popup views results in a significant increase in message size, this is not a significant concern here, with growing memory use by track trails a more substantial limiting factor.

A local deployment may recover from a clean start with 2 912 tracks/rotation (154 kB/s) with a 4-second scanning radar and 1 472 tracks/scan (312 kB/s) with a 1-second tracking radar with rendering enabled.

The cloud-based deployment reached 2 208 tracks/rotation (117 kB/s) for the 4-second scanning radar case and 320 tracks/scan (68 kB/s) for the 1-second tracking radar case, with both network throughput and latency constraints being bottlenecks here. Higher rates result in an excessive outgoing queue build up in the subsystem, with the scene becoming out of sync with the current data.

Sudden track bursts are often associated with suboptimal system behaviour caused by an incorrect setup, false data, or sudden increases in clutter. When a large number of low-quality tracks are added, they tend to be short-lived; such surges should be handled gracefully so that high-quality tracks are not impacted.

As long as the browser-based application can keep up with incoming data for either the scanning or tracking radar case, the same mean trail length constraints as indicated in figure 6.9 applies. This shows that the system is capable of handling a broad variety of track injection scenarios, including both low persistent and high burst injection.

The processing of intricate messages has a high processing cost on the browser-based HMI, which may result in the application becoming overwhelmed. With track updates being less expensive than creation, sudden excessive continuous track injections may cause the same cascading behaviour as seen with strobes (see section 6.2.2.2).

Track cleanup does result in increased CPU consumption since clean-out is handled individually, and each track consists of several DOM elements. Since not all track operations are staggered, performing cleanup processes with a large number of active tracks may result in intermittent drops in HMI responsiveness.

## 6.3 Summary

The general usability, versatility and performance of the proposed framework and its accompanying browser-based HMI, using representative signal chains, is summarized here.

Web applications allow for general cross-platform HMI client use, with functionality demonstrated on desktop and small-screen mobile devices. The most extensive use is for subsystem control and tracked data elements; however, if new visual elements are created at regular intervals, in bulk, mobile devices may exhibit non-ideal behaviour.

PWAs are also demonstrated in server-less use, with widespread support indicated across all platforms and sub-second HMI loading times determined where pre-cached.

The framework is then used to join subsystems dynamically into single-flow signal chains. These signal chains encompass a wide gamut of high-level uses, including system control and data recording, as well as data presentation from individual sources, through to fully implemented signal chains with numerous processing elements. Subsystems implemented with Python and Matlab locally and Python on an Azure-based VM are also confirmed.

The performance constraints with the framework on local and remote general-purpose hardware are emphasized. A signal chain with a single Spoofer subsystem injects plot, strobe and track data records to the HMI in packed payload blocks. For acceptable use, updates should not fall behind over time, the system should remain stable, and the HMI stay reasonably responsive to operator input. The rates shown in this section are determined as achievable for consistent throughput, as well as limited burst conditions.

Table 6.1 shows the upper-performance limits for non-interactive plots and clutter points, fluctuating around these values by up to ~20 000 plot/clutter points.

TABLE 6.1

High-level plot (50 B record) and clutter (21 B record) throughput

| | **Local** | **Remote** |
|---|---|---|
| Plot Points | 536 000 (6.70 MB/s) | 2 512 (125 kB/s) |
| Clutter Points | 768 000 (4.03 MB/s) | 20 976 (110 kB/s) |

Clutter throughput, in particular, offers a limited view, as a fully actualized implementation would benefit from a direct feed-through of raster images. Such would allow for individual radar resolution cells to be directly represented on the map, presenting in a more familiar way, instead of as a heatmap that requires image processing at the subsystem.

Table 6.2 shows the upper-performance limits for non-interactive strobes that are tracked and updated in consecutive scans through a unique identifier. Higher rates are achievable where excess strobes can still be formed before the initially created strobes time out.

TABLE 6.2

High-level strobe (63 B record) throughput

| | **Local** | **Remote** |
|---|---|---|
| Strobes | 5 120 (80.64 kB/s) | 5 120 (80.64 kB/s) |

Table 6.3 shows the upper-performance limits for interactive track updates spread over a 4 second period and presented in bulk in 1 second intervals. Higher rates are achievable where track injection rates ramp up over a period.

TABLE 6.3

High-level track (212 B record) throughput

| | **Local** | **Remote** |
|---|---|---|
| Scanning (4 s) | 2 912 (154 kB/s) | 2 208 (117 kB/s) |
| Tracking (1 s) | 1 472 (312 kB/s) | 320 (68 kB/s) |

Due to the increased memory requirement of extending track trails, HMI responsiveness is also affected by the duration that tracks are present. It respectively results in a display of at most 290 and 160 track updates for the scanning and tracking cases presented in table 6.3. The achievable track length to a specific input rate for a particular injection rate roughly follows an inverse power curve (see figure 6.9).

# Chapter 7

# Conclusion

Through an integrated framework and web application, this study investigated the incorporation of modern internet technologies in the development of operator-facing radar systems.

At first glance, internet technologies appear to be an unusual fit for radar systems, given their often significant throughput, latency, and processing requirements. However, with intensifying use of high-level software-based modules, increased levels of abstraction, and general domain innovations, it's becoming a very appealing option worth exploring further.

This appeal is broadly due to modern internet technologies, such as IoT, allowing for highly scalable dynamic systems through self-describing structures and the relative ease of integrating supplemental features. To most effectively determine the degree to which these features translate to radar systems development, a functional prototype is used to experimentally verify general usage and performance characteristics as they apply to specific usage scenarios.

It reveals the framework's approach as a flexible means to create high-level radar systems with low NRT demands by enabling a wide range of basic single-flow signal chain configurations from a small number of fundamental component types.

Subsequent sections provide a review of findings on the versatility and broad performance characteristics afforded by the technologies under consideration for radar development. Finally, some recommendations for future work, as follows from this study, are provided.

## 7.1 Review of findings

The research questions are defined to specify internet-technologies of note, the type of radar applications that are achievable using these technologies and to what degree the flexibility common to the web domain translate to radar system use as from sections 3.3 and 3.4.

### 7.1.1 Technologies of note

IoT messaging enables selective data access and a high degree of propagation, both desirable qualities within a signal chain. WebSockets-enabled MQTT enables multi-tenancy through a shared data model independent of any web server. It does provide lower throughput than other protocols but is widely supported and offers good tooling options (see section 2.3.4.2).

When high-availability access or a distributed solution is required, cloud services provide a complete solution (see section 6.1.2.6). These cloud-based solutions have relatively constrained throughput and latency, increasing the likelihood of delayed control messages and out-of-sync data flows, restricting their use where real-time processing algorithms are required (see section 6.2.2). Cloud-based usage also imposes some service level constraints and necessitates secure connections, creating additional overheads. Given these factors, cloud-based solutions benefit from having the most abstract map overlays viable and keeping parts of the signal chain that require extensive data transfers within the cloud, where direct access to various platforms and data services is available.

When used in conjunction with a web application, the approach allows for cross-platform client access across a wide range of devices. PWA technology also decouples the application from the need for an always-available web server by retaining the required HMI logic and resources on the browser, allowing consistent sub-second startup times (see section 6.8).

### 7.1.2 Viable applications

Specific viability is particularly determined from the throughput performance of high-level data and the nature of bottlenecks. Several usage scenarios were considered, including the direct display of a data stream, as well as recording and playback of archived data.

The types of systems that can be deemed viable depend predominantly on the performance of radar data throughput, such as with plots, tracks and strobes in section 6.2.2, with control messaging being far more restrained in comparison.

Here the system offers favourable low-persisted to the high-burst ratio for NRT track injections, with 50 persisted tracks kept for over 2 hours and the absorption of single scan bursts of near 3 000 tracks reached (see figure 6.9). Transfer rates approaching the MQTT/Websockets throughput limit (determined at about 7 MB/s from preliminary testing) are possible with small, efficiently packed plot messages that require minimal client-side processing. As such, protocol constraints are not a significant concern where client-side processing is the primary bottleneck, as with current-generation systems.

Web applications suffer decreased responsiveness with high data throughput or element count. Even so, it is not prohibitive to high-level, limited-area use, where it provides adequate performance when considering levels of visual clutter and practical usage.

### 7.1.3 Usage flexibility

This investigation showed that a highly adaptable data model is achievable by expressing it in one of the flexible data interchange formats common to web-based environments. Each subsystem's data model is, in turn, described in human-readable configuration files (as widely referenced in section 6.1.2). Before any changes to these files become reflected, the relevant subsystems must be restarted and maybe even re-added to the signal chain that integrates them. Nonetheless, it represents a significant advance in system adaptability and dynamic scalability over traditional radar system implementations.

Current programming languages widely support IoT application-level protocol use, which is advantageous for monolingual implementations of particular subsystems. However, since memory maps enable cross-language use, with a central assembly then providing control and supervision of that subsystem, this is not a strict requirement (see section 6.1.2.2).

Modern internet technologies, especially those related to the IoT, are in their early stages but already offer a suitable solution to many radar system needs. Future improvements are of distinct interest as efforts to standardise distributed network protocols are anticipated to result in more efficient data transfers, decreased latencies, and improved interoperability.

## 7.2 Future work

Future improvements would likely involve an expanded structure to allow for broader signal chains. Additionally, certain HMI additions, such as WebGL projected imagery for fine-grained radar maps, would be worthwhile.

Weather radar data may still be added in the future owing to the implementation's flexible design. To that aim, Matlab on the cloud would be useful when analysing these raw data archives, especially with its ability to interface with cloud-based HPC services; however, this involves extra costs and requires specific logic beyond this study's immediate scope.

Tracks are unusual in that they result in a substantial build-up of memory. A configurable trail length limit will allow the subsystem developer to set the track duration and counts as anticipated and with the help of figure 6.9.

Although using Azure's MQTT broker service requires custom logic, it would still be worth investigating further as a high-availability managed messaging option.

# References


[1] T. Debatty, "Software defined radar a state of the art," pp. 253–257, 06 2010.

[2] T. Grydeland, F. Lind, P. Erickson, and J. Holt, "Software radar signal processing," *Annales Geophysicae*, vol. 23, 01 2005.

[3] N. C. for Environmental Information. (2020) Radar data in the noaa big data project. website. Accessed May 2020. [Online]. Available: https://www.ncdc.noaa.gov/data-access/radar-data/noaa-big-data-project

[4] W. A. H. Mark A. Richards, James A. Scheer, *Principles of Modern Radar - Basic Principles*, 2nd ed. Edison, NJ: Scitech Publishing, 2010.

[5] Sajay K R and S. S. Babu, "A study of cloud computing environments for high performance applications," in *2016 International Conference on Data Mining and Advanced Computing (SAPIENCE)*, 2016, pp. 353–359.

[6] K. Jamil, M. Alam, M. Hadi, and Z. Alhekail, "A multi-band multi-beam software-defined passive radar. part i: System design," vol. 2012, 01 2012, pp. 1–4.

[7] J. Wang, M. Pierce, Y. Ma, G. Fox, A. Donnellan, J. Parker, and M. Glasscoe, "Using service-based gis to support earthquake research and disaster response," *Computing in Science & Engineering*, vol. 14, pp. 21–30, 09 2012.

[8] P. Kumar and B. Dezfouli, "Implementation and analysis of quic for mqtt," *Computer Networks*, vol. 150, 12 2018.

[9] R. G. A. Banks, "Mqtt version 3.1.1 plus errata 01," OASIS Open, Standard, December 2015, accessed April 2020. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html

[10] "Dod design criteria standard - human engineering," U.S. Army Aviation and Missile Command, Redstone Arsenal, AL, Standard, January 2012.

[11] "Nasa human integration design handbook," National Aeronautics and Space Administration, Standard, June 2014, accessed April 2020. [Online]. Available: https://www.nasa.gov/sites/default/files/atoms/files/human_integration_design_handbook_revision_1.pdf

[12] Y. Lim, A. Gardi, R. Sabatini, S. Ramasamy, and R. Bolia, "Avionics human-machine interfaces and interactions for manned and unmanned aircraft," *Progress in Aerospace Sciences*, 08 2018.

[13] "Us army weapon systems human-computer interface style guide, version 3," U.S. Department of the Army, Richland, WA, Standard, December 1999.

[14] "Faa nexgen business case letter," U.S. Department of Transportation, FAA, Washington, DC, Business Case, August 2017.

[15] A. Kluber. (2017, January) Faa seeks consolidation for non-cooperative air surveillance radars. web. Accessed May 2020. [Online]. Available: https://www.aviationtoday.com/2017/01/06/faa-seeks-consolidation-for-non-cooperative-air-surveillance-radars/

[16] T. Lewis, K. Burke, M. Underwood, and D. Wing, "Weather design considerations for the tasar traffic aware planner," 06 2019.

[17] A. L. Galina Havin, Larry Arend. (2020) Design of air traffic management graphics (example). web. Accessed April 2020. [Online]. Available: https://colorusage.arc.nasa.gov/ATM_1.php

[18] L. V. David Marconnet, Christian Norden. (2016, July) Optimum use of weather radar.

[19] "United states standard flight inspection manual," US Department of the Army, the Navy, and the Air Force and the Federal Aviation Administration, Washington, DC, Manual, October 2005.

[20] D. Sunday, T. Barrett, M. Dennis, C. Frangos, M. Schlegel, and T. Whitaker, "E-2c hawkeye combat system display," vol. 23, pp. 209–222, 04 2002.

[21] E. Lee, S. Kim, and Y. Kwon, "Analysis of interface and screen for ground control system," *Journal of Computer and Communications*, vol. 04, pp. 60–65, 01 2016.

[22] "The csir dossier," Council for Scientific and Industrial Research, ZA, Pretoria, ZA, Publication, September 2016, accessed April 2020. [Online]. Available: https://www.csir.co.za/sites/default/files/Documents/Dossier_Aug2016_Draft8_final%20lowres%20file.pdf

[23] O. Hüppop, M. Ciach, R. Diehl, D. Reynolds, P. Stepanian, and M. Menz, "Perspectives and challenges for the use of radar in biological conservation," *Ecography*, 12 2018.

[24] B. Eged, R. Fiengo, and D. Hall, "Modular platform approach for uwb radar system design and verification challenges," *Microwave Journal*, pp. 8–14, 08 2014.

[25] D. Barton, "Development of the an/fps-16 instrumentation radar," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 26, pp. B1–B16, 04 2011.

[26] F. Neri, *Introduction to Electronic Defence Systems*, 2nd ed. Edison, NJ: Scitech Publishing, 2006.

[27] E. F. Knott, *Radar Handbook, Radar Cross Section.* Pennsylvania, NY: McGraw-Hill Education, 2008.

[28] M. Sagayarap, V. Jithesh, J. Singlr, D. Roshanr, and K. Srinivasa, "A hybrid approach to cognition in radars," *Defence Science Journal*, vol. 68, pp. 183–189, 03 2018.

[29] D. Oikonomou, P. Nomikos, G. Limnaios, and K. Zikidis, "Passive radars and their use in the modern battlefield," vol. 9, pp. 37–61, 04 2019.

[30] A. Cockburn. (2005) Hexagonal architecture. Accessed March 2022. [Online]. Available: https://alistair.cockburn.us/hexagonal-architecture/

[31] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, A System of Patterns*, 1st ed. Chichester, West Sussex, England: John Wiley and Sons Ltd, 1996.

[32] L. A. Haws, "Baseband digital signal processing of radar system," 2015.

[33] V. Emeakaroha, K. Fatema, P. Healy, and J. Morrison, "Towards a generic cloud-based sensor data management platform: A survey and conceptual architecture," *SENSORCOMM 2014 - 8th International Conference on Sensor Technologies and Applications*, pp. 88–95, 01 2014.

[34] G. Gracioli, M. Dunne, and S. Fischmeister, "A comparison of data streaming frameworks for anomaly detection in embedded systems," *1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec)*, 04 2018.

[35] U. of Southern California, "Internet protocol - darpa internet program - protocol specification," University of Southern California, Marina del Rey, CA, Standard, September 1981.

[36] "Transmission control protocol - darpa internet program - protocol specification," University of Southern California, Marina del Rey, CA, Standard, September 1981.

[37] "Udp usage guidelines," rfc, Internet Engineering Task Force, Usage Guidelines, March 2017.

[38] J. Dizdarevic, F. Carpio, A. Jukan, and X. Masip, "A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration," *ACM Computing Surveys*, vol. 51, 04 2018.

[39] D. Evans, "The internet of things how the next evolution of the internet is changing everything," Cisco, San Jose, CA, Whitepaper, April 2011.

[40] S. Celar, E. Mudnic, and Z. Seremet, *State-Of-The-Art of Messaging for Distributed Computing Systems*, 01 2016, pp. 0298–0307.

[41] P. Cui, "Comparison of iot application layer protocols," 2017.

[42] C. Gündoğran, P. Kietzmann, M. Lenders, H. Petersen, T. C. Schmidt, and M. Wählisch, "Ndn, coap, and mqtt a comparative measurement study in the iot," *Proceedings of the 5th ACM Conference on Information-Centric Networking - ICN '18*, 2018. [Online]. Available: http://dx.doi.org/10.1145/3267955.3267967

[43] M. Amaran, N. Noh, and H. Hashim, "A comparison of lightweight communication protocols in robotic applications," *Procedia Computer Science*, vol. 76, pp. 400–405, 12 2015.

[44] mosquitto MQTT. (2020) mosquitto.conf man page. Accessed April 2020. [Online]. Available: https://mosquitto.org/man/mosquitto-conf-5.html

[45] T. H. Team. (2015) Last will and testament - mqtt essentials: Part 9. Accessed September 2021. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament/#:~:text=In%20MQTT%2C%20you%20use%20the%20Last%20Will%20and,a%20topic%2C%20retained%20message%20flag%2C%20QoS%2C%20and%20payload.

[46] J. Barabas. (2020) Iaas, paas and saas – ibm cloud service models. web. Accessed April 2020. [Online]. Available: https://www.ibm.com/za-en/cloud/learn/iaas-paas-saas

[47] J. Naren, S. Sowmya, and P. Deepika, "Layers of cloud – iaas, paas and saas: A survey," *International Journal of Computer Science and Information Technology*, vol. Vol. 5 (3), pp. 4477 – 4480, 06 2014, accessed April 2020. [Online]. Available: https://www.researchgate.net/publication/264458816_Layers_of_Cloud_-_IaaS_PaaS_and_SaaS_A_Survey

[48] C. Kotas, T. Naughton, and N. Imam, "A comparison of amazon web services and microsoft azure cloud platforms for high performance computing," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, 2018, pp. 1–4.

[49] T. Ludescher, T. Feilhauer, and P. Brezany, "Cloud-based code execution framework for scientific problem solving environments," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2, p. 11, 01 2013.

[50] E. Kosenko, "Industrial IoT Development for Condition Monitoring Purposes," Lappeenranta, Finland, 2018, masters Thesis.

[51] "Matlab distributed computing server cloud center user's," The MathWorks, Inc., Natick, MA, User Manual, December 2015.

[52] T. Pessoa, R. Medeiros, T. Nepomuceno, G.-B. Bian, V. Albuquerque, and P. P. Filho, "Performance analysis of google colaboratory as a tool for accelerating deep learning applications," *IEEE Access*, vol. PP, pp. 1–1, 10 2018.

[53] Matlab. (2020) Matlab online. Accessed June 2020. [Online]. Available: https://www.mathworks.com/products/matlab-online.html

[54] C. Reed, L. Mckee, and S. Ramage, "Ogc standards and cloud computing," 11 2010.

[55] O. Terzo, P. Ruiu, E. Bucci, and F. Xhafa, "Data as a service (daas) for sharing and processing of large data collections in the cloud," 07 2013, pp. 475–480.

[56] R. Saturi, S. Saturi, and P. Reddy, "Data as a service (daas) in cloud computing [data-as-a-service in the age of data] data as a service daas in cloud computing data-as-a-service in the age of data data as a service (daas) in cloud computing [data-as-a-service in the age of data]," vol. 12, 01 2012.

[57] (2020) Mqtt basics. web. Natick, MA. Accessed May 2020. [Online]. Available: https://www.mathworks.com/help/matlab/import_export/work-with-remote-data.html

[58] B. Scully and K. Mitchell, "Archival automatic identification system (ais) data for navigation project performance evaluation," 08 2015.

[59] S. Mao, E. Tu, G. Zhang, L. Rachmawati, E. Rajabally, and G.-B. Huang, "An automatic identification system (ais) database for maritime trajectory prediction and data mining," 07 2016.

[60] A. Hub. (2020) Ais hub - ais data sharing and vessel tracking. Accessed April 2020. [Online]. Available: http://www.aishub.net/

[61] J. Zhang, W. LIU, and Y. ZHU, "Study of ads-b data evaluation," *Chinese Journal of Aeronautics - CHIN J AERONAUT*, vol. 24, pp. 461–466, 08 2011.

[62] O. Network. (2020) Open air traffic data for research. Accessed April 2020. [Online]. Available: https://opensky-network.org/

[63] K. hu, Z. Gui, X. Cheng, H. Wu, and S. McClure, "The concept and technologies of quality of geographic information service: Improving user experience of giservices in a distributed computing environment," *International Journal of Geo-Information*, vol. 8, 03 2019.

[64] "Opengis® web map server implementation specification version 1.3," Open Geospatial Consortium, Wayland, MA, specification, March 2006.

[65] H. Shah and T. Soomro, "Node.js challenges in implementation," 05 2017.

[66] J. H. B. Terlson, B. Farias, "Ecmascript 2019 language specification," European Computer Manufacturers Association, Geneva, CH, Specification, June 2019.

[67] T. A. Majchrzak, A. Biørn-Hansen, and T.-M. Grønli, "Progressive web apps: the definite approach to cross-platform development?" in *HICSS*, 2018.

[68] M. Yeh, C. Swider, Y. Jo, and C. Donovan, "Human factors considerations in the design and evaluation of flight deck displays and controls, version 2.0," Tech. Rep., 10 2019.

[69] Google. (2020) Material foundation - color - dark theme. web. Accessed May 2020. [Online]. Available: https://material.io/design/color/dark-theme.html#usage

[70] "Dod interface standard - joint military symbology," U.S. Army Aviation and Missile Command, Redstone Arsenal, AL, Standard, November 2014.

[71] A. L. Galina Havin, Larry Arend. (2010) Individual differences in color vision. web. Accessed April 2020. [Online]. Available: https://colorusage.arc.nasa.gov/indiv_diffs.php

[72] L. Bollini, "Beautiful interfaces - from user experience to user interface design," *The Design Journal*, vol. 20, no. sup1, pp. S89–S101, 2017, accessed April 2020. [Online]. Available: https://doi.org/10.1080/14606925.2017.1352649

[73] A. Cockburn, C. Gutwin, and S. Greenberg, "A predictive model of menu performance," 01 2007, pp. 627–636.

[74] X. Bi, Y. Li, and S. Zhai, "Ffitts law: Modeling finger touch with fitts' law," 04 2013, pp. 1363–1372.

[75] Google. (2020) Material design - elevation. web. Accessed April 2020. [Online]. Available: https://material.io/design/environment/elevation.html

[76] D. Skvorc, M. Horvat, and S. Srbljic, "Performance evaluation of websocket protocol for implementation of full-duplex web streams," 05 2014, pp. 1003–1008.

[77] A. Deveria. (2020) Can i use - websockets, bidirectional communication technology for web apps. Accessed April 2020. [Online]. Available: https://caniuse.com/#feat=websockets

[78] (2020) Eclipse paho mqtt c client. repo. Accessed April 2020. [Online]. Available: https://github.com/eclipse/paho.mqtt.c

[79] (2020) Eclipse paho mqtt c++ client library. repo. Accessed April 2020. [Online]. Available: https://github.com/eclipse/paho.mqtt.cpp

[80] F. P. R. Light. (2019) paho-mqtt 1.5.0. package library. Accessed April 2020. [Online]. Available: https://pypi.org/project/paho-mqtt/

[81] R. Weilbacher. (2020) Mqtt.jl. repo. Accessed April 2020. [Online]. Available: https://github.com/rweilbacher/MQTT.jl

[82] (2020) Mqtt basics. web. Natick, MA. Accessed April 2020. [Online]. Available: https://www.mathworks.com/help/thingspeak/mqtt-basics.html

[83] I. d. N. O. Ben-Kiki, C. Evans, "Yaml version 1.2 edition 03," Specification, October 2009, accessed August 2021. [Online]. Available: https://yaml.org/spec/1.2/spec.pdf

[84] "Nato joint military symbology app-6(c)," NATO Standardization Agency, Brussels, BE, Standard, May 2011.

[85] Mozilla. (2021, January) Remove the ssb feature. Accessed December 2021. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1682593

[86] Google. (2021, November) Progressive web apps. Accessed March 2022. [Online]. Available: https://web.dev/learn/pwa/progressive-web-apps/

[87] "Html living standard," WHATWG, W3C, Standard, April 2020, accessed April 2020. [Online]. Available: https://html.spec.whatwg.org/print.pdf

[88] J. Ndia, G. Muketha, and K. Omieno, "Complexity metrics for sassy cascading style sheets," *Baltic Journal of Modern Computing*, vol. 7, 01 2019.

[89] N. Weizenbaum. (2017, February) Sass and browser compatibility. blog. Accessed April 2020. [Online]. Available: https://sass-lang.com/blog/sass-and-browser-compatibility

[90] K. Chaturvedi, "Web based 3d analysis and visualization using html5 and webgl," 03 2014.

[91] "Scalable vector graphics (svg) 1.1 (second edition)," W3C, Standard, August 2011. [Online]. Available: https://www.w3.org/TR/SVG11/

[92] MDN. (2020, March) Webgl: 2d and 3d graphics for the web. rfc. Accessed June 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[93] V. Agafonkin. (2017, July) How i built a wind map with webgl. web. Accessed April 2020. [Online]. Available: https://blog.mapbox.com/how-i-built-a-wind-map-with-webgl-b63022b5537f

[94] J. R. R. Fielding. (2014, June) Hypertext transfer protocol (http/1.1): Semantics and content. rfc. Accessed April 2020. [Online]. Available: https://tools.ietf.org/html/rfc7231

[95] A. M. I. Fette. (2011, December) The websocket protocol. rfc. Accessed April 2020. [Online]. Available: https://tools.ietf.org/html/rfc6455

[96] E. T. M. Belshe, R. Peon. (2015, May) Hypertext transfer protocol version 2 (http/2). rfc. Accessed June 2020. [Online]. Available: https://tools.ietf.org/html/rfc7540

[97] P. McManus. (2018, September) Bootstrapping websockets with http/2. rfc. Accessed June 2020. [Online]. Available: https://tools.ietf.org/html/rfc8441

[98] D. Edler and M. Vetter, "The simplicity of modern audiovisual web cartography: An example with the open-source javascript library leaflet.js," *KN - Journal of Cartography and Geographic Information*, 02 2019.

[99] V. Agafonkin. (2019) Leaflet and open-source javascript library for mobile-friendly interactive maps. web. Accessed April 2020. [Online]. Available: https://leafletjs.com/

[100] G. Maggini. (2019, July) Deciding between native and cross-platform mobile frontend programming frameworks. web. Accessed September 2021. [Online]. Available: https://v4.webpack.js.org/concepts/why-webpack/#esm---ecmascript-modules

# Appendix A

# Technologies

## A.1 Transport Layer Protocols

### A.1.1 User Datagram Protocol (UDP)

This protocol has a packet header size of 8 B, with a payload limit of 65 507 B over IPv4 and 65 527 B over IPv6 [37]. UDP offers multi-casting but does not guarantee that data will reach a recipient. A tendency to drop packets under non-ideal network conditions places a greater burden on endpoints to ensure communications reliability.
UDP necessitates the use of loss tolerant algorithms or additional logic to ensure that data losses remain at reasonable levels and network receivers do not become over-saturated.[2]

### A.1.2 Transmission Control Protocol (TCP)

This protocol has a minimum packet header size of 20 B, with a payload limit of 65 535 B. [36] TCP is more reliable than UDP, but has significant overhead and uses the available bandwidth less efficiently. [2]
TCP offers flow control by giving the receiver some control over the send rate. Data is ordered through sequence numbers and each packet must be acknowledged by the receiver, allowing for the spanning of larger payloads over multiple packets. [36, p. 3-4]

## A.2 HTML5

HTML5 is a popular buzzword for the amalgamation of various technologies used to create web applications. These technologies are used to create a unified Document Object Model (DOM) composed of various nodes arranged in a tree-like structure. This DOM tree is made up of an "html" node at the top, followed by "head", "body" and "text" nodes. The "head" and "body" nodes are then used to build an in-memory representation of application resources, allowing for direct interaction through various APIs. [87, p. 24, 27, 30-31]

### A.2.1 HyperText Markup Language (HTML)

This abstract language is at the heart of the modern web, serving as the main syntax for describing DOM structures. It was originally designed to semantically define scientific documents, providing a media-independent description of interactive content. [87, p. 24]

HTML is primarily based on the Standard Generalized Markup Language (SGML) [87, p. 1059], with the same tag-matched appearance present as in other SGML profiles, such as XML. An HTML document contains a structure that closely mirrors the DOM it describes, with each element in the tree being denoted by a particular start and end tag. The start tag may contain various attributes of the associated DOM node. Tags representing child nodes in the DOM tree are nested between this start and end-tag, allowing for the description of very deep tree structures. [87, p. 24, 27, 30-31]

### A.2.2 Cascading Style Sheets (CSS)

This styling language allows for the centralized definition of styling rules, greatly assisting in the enforcement of a unified appearance. It provides control over certain DOM attributes that govern rendering, such as opacity, colour and stroke widths. [87, p. 3, 5, 105, 1172]

Since browsers apply styling rules differently, cross-browser validation is required at all stages of development. This can be mitigated by using a CSS pre-processor, such as Sassy CSS (SCSS) [88], to generate browser-specific styling rules [89] that ensure the desired appearance across all specified browsers and browser versions.

### A.2.3 ECMAScript

This is a general-purpose, object-oriented, weakly typed programming language derived from the JScript and JavaScript web scripting languages. It is syntactically based on Java, but with some changes to make it an easy-to-use scripting language. [66, p. 4-7] It appears to have inherited the JavaScript moniker, and the terms are currently used interchangeably in web application development. [87, p. 53]

ECMAScript is the preferred scripting language within HTML5 [87, p. 63] and is often used to effect web workers [87, p. 1009]. It allows for control over the DOM and can be directly embedded into HTML [87, p. 31]. Larger ECMAScript programs are often bundled as modules that may be loaded through HTML. [66, p. 6] [87, p. 1009]

Modern ECMAScript iterations are deemed suitable for use on web servers, with these providing the requisite site resources to connecting web clients [90]. It effectively distributes computing between a browser and a server, resulting in the complete host environment for ECMAScript programs. [66, p. 7]

### A.2.4 Scalar Vector Graphics (SVG)

This vector image format supports fully scalable visualisations, resulting in high-quality representations at a range of display resolutions. It is an HTML5 standard format that is widely supported in modern browsers. SVG is a scripting language in and of itself, and it can even be used to create fully functional web applications when used on its own. It is particularly useful for the development of standalone embedded components when coupled with other HTML5 technologies.

SVGs create their own DOM, providing access to its internal elements, attributes and properties. This allows for the styling of SVG elements to be done with CSS, as well as integration with external scripting languages. Any SVG graphical object can be assigned a large number of event handlers (e.g. ‘onmouseover’, ‘onclick’, etc.), allowing for dynamic interaction with specified control visualizations. As a result, it should be a suitable choice for creating typical control elements like sliders, buttons, toggles, and indicators. [91, p. 20, 29-32, 511, 755][87, p.442]

### A.2.5 Web Graphics Library (WebGL)

This JavaScript API allows for enhanced rendering by providing access to some of the graphics card's shader stages. WebGL is heavily influenced by OpenGL ES 2.0, which is a Khronos Group specification. [90, 92]

It functions by extending the HTML5 canvas element and is supported in most major modern web browsers. It is widely used in blue marble web applications that require complex visualisations (e.g. city models and animated terrain). [90, 93]

## A.3 Web-oriented Application Layer Protocols

### A.3.1 HyperText Transfer Protocol (HTTP)

This protocol is widely used for content negotiation over the internet, typically with TCP as the transfer. It is tightly interwoven with the 'Request-Response' based Representational State Transfer (REST) architectural pattern [94, p. 7]. This versatile pattern is premised on the manipulation and transfer of representations that reflect the past, present or future state of a given resource. It has become a mainstay for web service interaction, with universal support by both server and client-side drivers. [38, p. 7]

REST HTTP requests has metadata indicating how a payload should be interpreted [94, p. 8-18], ensuring a uniform interface through which to interact with type-less, potentially unbounded, data sources. The HTTP response contains a message field that makes use of a standardized list of status codes to indicate transaction success or specific failure conditions [94, p. 47-64].

HTTP shows performance benefits with larger payloads [38, p. 16], offering a reliable means through which to deliver file data. It is inherently unsuitable for live messaging though, as it does not allow for push notifications, has long header fields, and a high level of relative complexity [38, p. 8]. This places an undue burden on connecting systems to facilitate live communication and requires a lot of network resources.

### A.3.2 WebSockets

This protocol enables near real-time messaging between server and client entities over TCP by allowing for simultaneous, bi-directional push operations. [38] It is widely supported on modern browsers [87, p.988].

It primarily aims to provide a standardised method of communication between servers and browser-based applications that eliminates the need for frequent HTTP sessions by keeping the connection alive. Communication is restricted through a same-origin policy and the protocol makes use of an elaborate handshake to ensure that its use is restricted to servers that have opted-in. [95, p. 1,10-11].

Although HTTP/2 directly allows for server push operations [96], wide support for WebSockets has solidified its use for live messaging on the web [97]. The WebSockets protocol is not inherently designed for resource-constrained scenarios [38], but the combination of HTTP/2 with WebSockets promises broad performance improvements here [97]. With a standard HTTP 1.1. implementation the client performance in consuming plot and track messages may be affected, but is likely to scale with appropriate payload structuring, as would be expected with standard TCP underlying this application-level protocol [76, p. 4].

## A.4 Other frameworks & libraries

### A.4.1 Node.js

Node.js provides a full, distributed ECMAScript host environment [66, p. 7] between the client and web server. Node.js has demonstrated support for Web GIS interaction [98] and can be accessed via a variety of SaaS offerings [65, p. 75].

Node.js includes an ECMAScript runtime environment, allowing ECMAScript to be used outside of the browser. It is relatively lightweight, scales well and is considered an appropriate solution to web server development where IoT technologies are used. This is primarily due to its event-driven nature and asynchronous handling of input and output. [65, p. 74, 77] It is openly available and offers a rich development ecosystem through the Node.js Package Manager (npm), which has the additional benefit of greatly simplifying web server deployment. [65, p. 75-76] Node.js can be deployed on various modern operating systems, including Microsoft's Windows, Apple's macOS and most Linux distributions. [65, p. 74]

### A.4.2 Leaflet

Leaflet is a well-known open-source, mobile-friendly web cartography package. Its logic may be included as an ECMAScript module on self-deployed web servers, eliminating the need for internet access. It is suitable for use in a standards-based implementation as it allows for direct DOM access and adheres to HTML layout and CSS styling rules.

Leaflet allows for easy integration with open access DaaS providers and can be configured to retrieve map tiles from any standards-compliant WMS. Tiles are requested from the configured WMS based on the current bounded area, zoom level, and output resolution.

Map markers can be placed on the base map in specific locations and then moved around programmatically. These markers can be styled with custom images in either raster or vector format, and they support operator interaction as well as association with contextual elements such as pop-ups. Leaflet allows geo-referenced shapes to be drawn over map layers, with inherent support for polylines, polygons, and circles.

In addition to these fundamental features, there is a growing collection of openly available elements that can be added to further expand its capabilities. [98, 99]

### A.4.3 WebPack

Modern web development may result in intricate development hierarchies due to the use of a range of languages and the widespread reliance on a selection of publicly available modules and resources. While organising them into nested hierarchies alongside code simplifies development, resulting in a fragmented structure that is undesirable for deployed web applications. Such may cause page loading times to be slower, especially where the code is not transfer-optimized.

With a bundler, such as WebPack, the development and deployment environments may be considered separately. It analyses code structures from a predefined set of entry points to build a dependency graph. This graph is then used to unify the fragmented development side structures into a minimum equivalent structure, typically resulting in a single transfer-optimized module.

WebPack also ensures that all associated scripts and languages, including dependencies, are transpiled to a common output format where applicable. [100]

# Appendix B

# Subsystem Configuration Files

Subsystem-specific configuration files based on YAML. Each subsystem is tied to a unique GUID and should broadcast its existence to the central broker at the specified endpoint upon execution.

These configuration files should also contain the subsystem's output data and control schemas. If the associated subsystem is part of the active signal chain, its controls will then be accessible via the HMI's corresponding subsystem panel.

Custom configurations utilised in subsystem logic, but not part of the broader disseminated assembly, may also be specified here.

Listing B.1: YAML subsystem config example

```
# SETUP
uid: <SUBSYSTEM-GUID>
name: <SUBSYSTEM-NAME>
broker:
  ip: <BROKER-BROADCAST-DESTINATION-IP>
  port: <BROKER-BROADCAST-DESTINATION-PORT>
  useTls: false # true for MQTTS
# DATA OUT
dataSchema:
  # List of subsystem output definitions
# CONTROLS
controlSchema:
  # List of subsystem control definitions
# CUSTOM
# Subsystem specific configurations
```

## B.1 ADSB Tracker subsystem config

Listing B.2: YAML subsystem config example

```
# SETUP
uid: <ADSB-GUID>
name: ADSB
broker:
  ip: 127.0.0.1
  port: 1883
  useTls: false
# DATA OUT
dataSchema:
  - key: Tracks
    type: text/csv
    display: Track
    charset: UTF-8
    dataTypes: string_24,string_5,float,float,float,float,uint32,string_254
    header: Time,Identifier,Latitude_deg,Longitude_deg,Bearing_deg,Speed_ms,Type,Info
    trailLength: 500
    classifications:
      - label: default
        symbolIndex: 0
        paletteIndex: 5
    refreshPeriod: PT90S
# CONTROLS
controlSchema:
  - type: TextBox
    uid: <CONTROL-GUID-0>
    label: Latitude Min
    value: "-45"
  - type: TextBox
    uid: <CONTROL-GUID-1>
    label: Latitude Max
    value: "0"
  - type: TextBox
    uid: <CONTROL-GUID-2>
    label: Longitude Min
    value: "0"
  - type: TextBox
    uid: <CONTROL-GUID-3>
    label: Longitude Max
    value: "50"
# CUSTOM
adbsUrl: "https://opensky-network.org/api/states/all?lamin={0}&lomin={1}&lamax={2}&lomax={3}"
```

## B.2 Control subsystem config

Listing B.3: YAML subsystem config example

```
# SETUP
uid: <CONTROLLER-GUID>
name: Control
broker:
  ip: 127.0.0.1
  port: 1883
  useTls: false
# CONTROLS
controlSchema:
  # Print state in console
  - type: Radio
    uid: <CONTROL-GUID-0>
    label: Print to console
    selected: 0
    items:
      - 'Yes'
      - 'No'
  # Print state in console
  - type: Radio
    uid: <CONTROL-GUID-1>
    label: Write mechanism
    selected: 0
    items:
      - 'Event based'
      - 'Poll based'
  # TextBox control
  - type: TextBox
    uid: <CONTROL-GUID-2>
    label: Value Control
    value: "Test"
  # CheckBox control
  - type: CheckBox
    uid: <CONTROL-GUID-3>
    label: CheckBox Control
    items:
      - label: Item0
        isChecked: false
      - label: Item1
        isChecked: false
  # Radio control
  - type: Radio
    uid: <CONTROL-GUID-4>
    label: Radio Control
    selected: 1
    items:
      - Item0
      - Item1
      - Item2
```

```
# Slider control
- type: Slider
  uid: <CONTROL-GUID-5>
  label: Slider Control
  min: 0
  max: 10
  value: 0
```

## B.3 Data Feeder subsystem config

Listing B.4: YAML subsystem config example

```
# SETUP
uid: <DATA-FEEDER-GUID>
name: Data Feeder
broker:
  ip: 127.0.0.1
  port: 1883
  useTls: false
# DATA OUT
dataSchema:
  - key: Raw
    type: application/octet-stream
    dataTypes: uint64,float,float,float,float
    header: Time_ms,Range_m,Azimuth_deg,Speed_ms,Intensity
# CONTROLS
controlSchema:
  # Slider control
  - type: Slider
    uid: <CONTROL-GUID-0>
    label: Burst Interval [ms]
    min: 0
    max: 1000
    value: 1000
  # TextBox control
  - type: TextBox
    uid: <CONTROL-GUID-1>
    label: Input Directory
    value: "./"
```

## B.4 Emulator subsystem config

Listing B.5: YAML subsystem config example

```
# SETUP
uid: <EMULATOR-GUID>
name: Emulator
broker:
  ip: 127.0.0.1
  port: 1883
  useTls: false
# DATA OUT
dataSchema:
  - key: Raw
    type: application/octet-stream
    dataTypes: uint64,float,float,float,float
    header: Time_ms,Range_m,Azimuth_deg,Speed_ms,Intensity
# CONTROLS
controlSchema:
  # Clutter points slider control
  - type: Slider
    uid: <CONTROL-GUID-0>
    label: Clutter / Sector
    min: 0
    max: 3000
    value: 50
  # Valid track points slider control
  - type: Slider
    uid: <CONTROL-GUID-1>
    label: Tracks / Sector
    min: 0
    max: 10
    value: 1
```

## B.5 Processor subsystem config

Listing B.6: YAML subsystem config example

```
# SETUP
uid: <PROCESSOR-GUID>
name: Processor
broker:
  ip: 127.0.0.1
  port: 1883
  useTls: false
# DATA OUT
dataSchema:
  - key: ClutterMap
    type: text/csv
    display: HeatMap
    charset: UTF-8
    dataTypes: float,float,uint32
    header: Latitude_deg,Longitude_deg,Intensity
    dotSize: 100
    refreshPeriod: PT4S
  - key: Plots
    type: text/csv
    display: Plot
    charset: UTF-8
    dataTypes: uint64,float,float,float,float,float,uint32
    header: Time_ms,Latitude_deg,Longitude_deg,Range_m,Azimuth_rad,Speed_ms,Type
    classifications:
      - label: type0
        paletteIndex: 0
      - label: type1
        paletteIndex: 1
      - label: type2
        paletteIndex: 6
      - label: type3
        paletteIndex: 12
    refreshPeriod: PT4S
# CONTROLS
controlSchema:
  # Slider control
  - type: Slider
    uid: <CONTROL-GUID-0>
    label: Plots Threshold
    min: 1
    max: 20
    value: 16
```

## B.6 Recorder subsystem config

Listing B.7: YAML subsystem config example

```
# SETUP
uid: <RECORDER-GUID>
name: Recorder
broker:
  ip: 127.0.0.1
  port: 1883
  useTls: false
# CONTROLS
controlSchema:
  # TextBox control
  - type: TextBox
    uid: <CONTROL-GUID-0>
    label: Output Directory
    value: "./"
```

## B.7 Spoofer subsystem config

Listing B.8: YAML subsystem config example

```
# SETUP
uid: <SPOOFER-GUID>
name: Spoofer
broker:
  ip: 127.0.0.1
  port: 1883
  useTls: false
# DATA OUT
dataSchema:
  - key: ClutterMap
    type: text/csv
    display: HeatMap
    charset: UTF-8
    dataTypes: float,float,uint32
    header: Latitude_deg,Longitude_deg,Intensity
    dotSize: 100
    refreshPeriod: PT4S
  - key: Plots
    type: text/csv
    display: Plot
    charset: UTF-8
    dataTypes: uint64,float,float,float,float,float,uint32
    header: Time_ms,Latitude_deg,Longitude_deg,Range_m,Azimuth_rad,Speed_ms,Type
    classifications:
      - label: type0
        paletteIndex: 0
      - label: type1
        paletteIndex: 1
      - label: type2
        paletteIndex: 6
      - label: type3
        paletteIndex: 12
    refreshPeriod: PT4S
  - key: Strobes
    type: text/csv
    display: Strobe
    charset: UTF-8
    dataTypes: string,string,float,float,float,float,uint32
    header: Time,Identifier,Origin_Latitude_deg,Origin_Longitude_deg,Range_m,Azimuth_deg,Type
    classifications:
      - label: type0
        paletteIndex: 2
      - label: type1
        paletteIndex: 8
      - label: type2
        paletteIndex: 10
      - label: type3
        paletteIndex: 13
```

```
    refreshPeriod: PT4S
  - key: Tracks
    type: text/csv
    display: Track
    charset: UTF-8
    dataTypes: string_24,string_5,float,float,float,float,uint32,string_254
    header: Time,Identifier,Latitude_deg,Longitude_deg,Bearing_deg,Speed_ms,Type,Info
    trailLength: 3000
    classifications:
      - label: type0
        symbolIndex: 0
        paletteIndex: 2
      - label: type1
        symbolIndex: 1
        paletteIndex: 4
      - label: type2
        symbolIndex: 2
        paletteIndex: 13
    refreshPeriod: PT4S
# CONTROLS
controlSchema:
  # Interval - Slider control
  - type: Slider
    uid: <CONTROL-GUID-0>
    label: Burst Interval [ms]
    min: 1
    max: 1000
    value: 250
  # ClutterMap - Radio control
  - type: Radio
    uid: <CONTROL-GUID-1>
    label: Clutter send mode
    selected: 0
    items:
      - Default
      - Perf. Test
  # ClutterMap - Slider control
  - type: Slider
    uid: <CONTROL-GUID-2>
    label: ClutterMap Rate [#/sector]
    min: 0
    max: 65536
    value: 0
  # Plots - Radio control
  - type: Radio
    uid: <CONTROL-GUID-3>
    label: Plots send mode
    selected: 0
    items:
      - Default
      - Perf. Test
  # Plots - Slider control
```

```
- type: Slider
  uid: <CONTROL-GUID-4>
  label: Plots Rate [#/sector]
  min: 0
  max: 65536
  value: 0
# Strobes - Slider control
- type: Slider
  uid: <CONTROL-GUID-5>
  label: Strobes Rate [#/sector]
  min: 0
  max: 4096
  value: 0
# Tracks - Radio control
- type: Radio
  uid: <CONTROL-GUID-6>
  label: Tracks send mode
  selected: 0
  items:
    - Scanning (4Hz)
    - Tracking (1Hz)
# Scanning Tracks - Slider control
- type: Slider
  uid: <CONTROL-GUID-7>
  label: Tracks Rate (scanning) [#/sector]
  min: 0
  max: 512
  value: 0
```

## B.8 Tracker subsystem config

Listing B.9: YAML subsystem config example

```
# SETUP
uid: <TRACKER-GUID>
name: Tracker
broker:
  ip: 127.0.0.1
  port: 1883
  useTls: false
# DATA OUT
dataSchema:
  - key: Tracks
    type: text/csv
    display: Track
    charset: UTF-8
    dataTypes: string_24,string_5,float,float,float,float,uint32,string_254
    header: Time,Identifier,Latitude_deg,Longitude_deg,Bearing_deg,Speed_ms,Type,Info
    trailLength: 500
    classifications:
      - label: type0
        symbolIndex: 0
        paletteIndex: 9
      - label: type1
        symbolIndex: 1
        paletteIndex: 11
      - label: type2
        symbolIndex: 2
        paletteIndex: 13
    refreshPeriod: PT4S
# CONTROLS
controlSchema:
  # M-Value slider control
  - type: Slider
    uid: <CONTROl-GUID-0>
    label: Required detections
    min: 2
    max: 6
    value: 3
```

# Appendix C

# Signal Chain Data Models

Data model structures for the MQTT-broker-based examples in this document are presented here. Major sections detail the active signal chain, while others provide all available signal chains and subsystems on the broker (as shown in their configurations in Appendix B), with the listed elements keyed to specific GUIDs.

JSON is primarily used within the MQTT message payloads to describe the signal chains and their subsystems. Subsystem data records are presented as CSV body segments, with the headers described elsewhere in the persisted data model.

Listing C.1: Data model description

```
{
  "AvailableChains": [ /* Ordered list of configured chains on the central broker */ ],
  "AvailableSubSystems": [ /* Ordered list of subsystems that have broadcast to the central broker */ ],
  "SelectedChain": { "id": "<SUBSYSTEM-GUID>", "isRunning": <true/false> },
  "Chains": { // Collection of chain configurations to the system, \eg{}:
    "<CHAIN-GUID>": {
      "Setup": {
        "label": "<CHAIN-NAME>",
        "origin": { "latitude": <ORIGIN-LATITUDE>, "longitude": <ORIGIN-LONGITUDE> },
        "range": "<ACTIVE-SENSOR-RANGE>",
        "SubSystems": [ /* List of subsystem GUIDs in chain */ ]
      },
      "SubSystems": {
        "<SUBSYSTEM-GUID>": {
          "Outgoing": { /* Optional outgoing endpoint and topic details for publishing/streaming */ },
          "Incoming": { /* Optional incoming endpoint and topic details for subscription/streaming */ },
          "Controls": { /* Collection of subsystem control definitions */ },
          "Data": { /* Collection of subsystem output definitions */ }}}}}}
```

Relevant data models for the examples discussed in this document, as well as topics that may be used to directly access particular parts of the data model.

Listing C.2: Data model example of configured signal chains and subsystems

```
{
// AvailableChains
  "AvailableChains": ["<SPOOFING-CHAIN-GUID>", "<CONTROL-CHAIN-GUID>", "<FULL-CHAIN-GUID>",
    "<RECORDER-CHAIN-GUID>", "<RECORDER-CHAIN-GUID>", "<PLAYBACK-CHAIN-GUID>", "<VIEWER-CHAIN-GUID>"],
  "AvailableSubSystems": {
    "<SPOOFER-GUID>": {
// AvailableSubSystems/<SPOOFER-GUID>/Status
      "Status": "",
// AvailableSubSystems/<SPOOFER-GUID>/Definition
      "Definition": { "label": "Spoofer", "streams": ["ClutterMap", "Plots", "Strobes", "Tracks"] }
    },
    "<CONTROLLER-GUID>": {
// AvailableSubSystems/<CONTROLLER-GUID>/Status
      "Status": "",
// AvailableSubSystems/<CONTROLLER-GUID>/Definition
      "Definition": { "label": "Control", "streams": [] }
    },
    "<EMULATOR-GUID>": {
// AvailableSubSystems/<EMULATOR-GUID>/Status
      "Status": "",
// AvailableSubSystems/<EMULATOR-GUID>/Definition
      "Definition": { "label": "Emulator", "streams": ["Raw"] }
    },
    "<PROCESSOR-GUID>": {
// AvailableSubSystems/<PROCESSOR-GUID>/Status
      "Status": "",
// AvailableSubSystems/<PROCESSOR-GUID>/Definition
      "Definition": { "label": "Processor", "streams": ["ClutterMap", "Plots"] }
    },
    "<TRACKER-GUID>": {
// AvailableSubSystems/<TRACKER-GUID>/Status
      "Status": "",
// AvailableSubSystems/<TRACKER-GUID>/Definition
      "Definition": { "label": "Tracker", "streams": ["Tracks"] }
    },
    "<RECORDER-GUID>": {
// AvailableSubSystems/<RECORDER-GUID>/Status
      "Status": "",
// AvailableSubSystems/<RECORDER-GUID>/Definition
      "Definition": { "label": "Recorder", "streams": [] }
    },
    "<DATA-FEEDER-GUID>": {
// AvailableSubSystems/<DATA-FEEDER-GUID>/Status
      "Status": "",
// AvailableSubSystems/<DATA-FEEDER-GUID>/Definition
      "Definition": { "label": "Data Feeder", "streams": ["Raw"] }
    },
```

```
    "<ADSB-GUID>": {
// AvailableSubSystems/<ADSB-GUID>/Status
      "Status": "",
// AvailableSubSystems/<ADSB-GUID>/Definition
      "Definition": { "label": "ADSB", "streams": ["Tracks"] }
    }}}
```

# C.1 Viewer chain

## C.1.1 Data model - Persisted (setup & control)

Listing C.3: Data model example (persisted)

```
{
// SelectedChain
  "SelectedChain": { "id": "<VIEWER-CHAIN-GUID>", "isRunning": true },
  "Chains": {
    "<VIEWER-CHAIN-GUID>": {
// Chains/<VIEWER-CHAIN-GUID>/Setup
      "Setup": {
        "label": "ADS-B",
        "origin": { "latitude": -33.971242, "longitude": 18.587132 },
        "range": "30000",
// Chains/<VIEWER-CHAIN-GUID>/Setup/SubSystems
        "SubSystems": ["<ADSB-GUID>"]
      },
      "SubSystems": {
        "<ADSB-GUID>": {
// Chains/<VIEWER-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Outgoing
          "Outgoing": { "protocol": "MQTT", "ip": "127.0.0.1", "port": 1883, "topics": ["Tracks"] },
// Chains/<VIEWER-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Incoming
          "Incoming": {},
          "Controls": {
// Chains/<VIEWER-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Controls/<CONTROL-GUID-0>
            "<CONTROL-GUID-0>": { "type": "TextBox", "label": "Latitude Min", "value": "-36" },
// Chains/<VIEWER-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Controls/<CONTROL-GUID-1>
            "<CONTROL-GUID-1>": { "type": "TextBox", "label": "Latitude Max", "value": "-20" },
// Chains/<VIEWER-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Controls/<CONTROL-GUID-2>
            "<CONTROL-GUID-2>": { "type": "TextBox", "label": "Longitude Min", "value": "15" },
// Chains/<VIEWER-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Controls/<CONTROL-GUID-3>
            "<CONTROL-GUID-3>": { "type": "TextBox", "label": "Longitude Max", "value": "36" }
          },
          "Data": {
            "Tracks": {
// Chains/<VIEWER-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Data/Tracks/Interpretation
              "Interpretation": { "key": "Tracks", "type": "text/csv", "display": "Track",
                "charset": "UTF-8", "trailLength": 500, "refreshPeriod": "PT90S"
```

```
                "dataTypes": "string_24,string_5,float,float,float,float,uint32,string_254",
                "header": "Time,Identifier,Latitude_deg,Longitude_deg,Bearing_deg,Speed_ms,Type,Info",
                "classifications": [
                  { "label": "default", "symbolIndex": 0, "paletteIndex": 5 }]}}}}}}}}
```

## C.1.2 Data model - Non-persisted (data)

### C.1.2.1 Data model

Listing C.4: Data model example (non-persisted)

```
{
  "AvailableSubSystems": {
    "<ADSB-GUID>": {
// AvailableSubSystems/<ADSB-GUID>/Status
      "Status": "Operational"
    }
  },
  "Chains": {
    "<VIEWER-CHAIN-GUID>": {
      "SubSystems": {
        "<ADSB-GUID>": {
// Chains/<VIEWER-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Rates
          "Rates": { "total": "000005", "errors": "000000" },
          "Data": {
            "Tracks": {
// Chains/<VIEWER-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Data/Tracks/Records
              "Records": /* ADSB-TRACK-RECORDS */ }}}}}}
```

### C.1.2.2 Records

Listing C.5: "<ADSB-GUID>/Data/Tracks/Records" example

```
"2021-09-18T11:10:55.257Z","00c458",-33.9779,18.605,267.19,7.2,0,
    "<b>CAW324A </b> (None)<br/><em>South Africa</em><hr/>ALT: 0 m<br/>C/R: <p style='...
"2021-09-18T11:10:55.257Z","00c452",-33.7389,18.7913,212.47,160.98,0,
    "<b>CAW453  </b> (None)<br/><em>South Africa</em><hr/>ALT: 1386 m<br/>C/R: <p style='...
"2021-09-18T11:10:55.257Z","00c453",-33.841,18.9725,50.85,178.45,0,
    "<b>CAW106  </b> (None)<br/><em>South Africa</em><hr/>ALT: 3642 m<br/>C/R: <p style='...
```

## C.2 Dedicated control chain

### C.2.1 Data model - Persisted (setup & control)

Listing C.6: Data model example (persisted)

```
{
// SelectedChain
  "SelectedChain": { "id": "<CONTROL-CHAIN-GUID>", "isRunning": true },
  "Chains": {
    "<CONTROL-CHAIN-GUID>": {
// Chains/<CONTROL-CHAIN-GUID>/Setup
      "Setup": {
        "label": "Controller",
        "origin": { "latitude": -33.9581, "longitude": 18.4601 },
        "range": "15000",
// Chains/<CONTROL-CHAIN-GUID>/Setup/SubSystems
        "SubSystems": ["<CONTROLLER-GUID>"]
      },
      "SubSystems": {
        "<CONTROLLER-GUID>": {
// Chains/<CONTROL-CHAIN-GUID>/SubSystems/<CONTROLLER-GUID>/Outgoing
          "Outgoing": { "protocol": "MQTT", "ip": "127.0.0.1", "port": 1883, "topics": [] },
// Chains/<CONTROL-CHAIN-GUID>/SubSystems/<CONTROLLER-GUID>/Incoming
          "Incoming": {},
          "Controls": {
// Chains/<CONTROL-CHAIN-GUID>/SubSystems/<CONTROLLER-GUID>/Controls/<CONTROL-GUID-0>
            "<CONTROL-GUID-0>": { "type": "Radio", "label": "Print to console", "selected": "0",
                                  "items": ["Yes", "No"] },
// Chains/<CONTROL-CHAIN-GUID>/SubSystems/<CONTROLLER-GUID>/Controls/<CONTROL-GUID-1>
            "<CONTROL-GUID-1>": { "type": "Radio", "label": "Write mechanism", "selected": "0",
                                  "items": ["Event based", "Poll based"] },
// Chains/<CONTROL-CHAIN-GUID>/SubSystems/<CONTROLLER-GUID>/Controls/<CONTROL-GUID-2>
            "<CONTROL-GUID-2>": { "type": "TextBox", "label": "Value Control", "value": "Test" },
// Chains/<CONTROL-CHAIN-GUID>/SubSystems/<CONTROLLER-GUID>/Controls/<CONTROL-GUID-3>
            "<CONTROL-GUID-3>": { "type": "CheckBox", "label": "CheckBox Control",
              "items": [
                { "label": "Item0", "isChecked": false },
                { "label": "Item1", "isChecked": false }
              ]
            },
// Chains/<CONTROL-CHAIN-GUID>/SubSystems/<CONTROLLER-GUID>/Controls/<CONTROL-GUID-4>
            "<CONTROL-GUID-4>": { "type": "Radio", "label": "Radio Control",
                                  "selected": "2", "items": ["Item0", "Item1", "Item2"] },
// Chains/<CONTROL-CHAIN-GUID>/SubSystems/<CONTROLLER-GUID>/Controls/<CONTROL-GUID-5>
            "<CONTROL-GUID-5>": { "type": "Slider", "label": "Slider Control",
                                  "min": 0, "max": 10, "value": "7" }}}}}}}
```

## C.2.2 Data model - Non-persisted (data)

### C.2.2.1 Data data model

Listing C.7: Data model example (non-persisted)

```
{
  "AvailableSubSystems": {
    "<CONTROL-GUID>": {
// AvailableSubSystems/<CONTROL-GUID>/Status
      "Status": "Operational"
    }
  },
  "Chains": {
    "<CONTROLLER-CHAIN-GUID>": {
      "SubSystems": {
        "<CONTROL-GUID>": {
// Chains/<CONTROLLER-CHAIN-GUID>/SubSystems/<CONTROL-GUID>/Rates
          "Rates": { "total": "000000", "errors": "000000" }}}}}}
```

# C.3 Recording chain

## C.3.1 Data model - Persisted (setup & control)

Listing C.8: Data model example (persisted)

```
{
// SelectedChain
  "SelectedChain": { "id": "<RECORDER-CHAIN-GUID>", "isRunning": true },
  "Chains": {
    "<RECORDER-CHAIN-GUID>": {
// Chains/<RECORDER-CHAIN-GUID>/Setup
      "Setup": {
        "label": "Recorder",
        "origin": { "latitude": -33.9581, "longitude": 18.4601 },
        "range": "15000",
    // Chains/<RECORDER-CHAIN-GUID>/Setup/SubSystems
        "SubSystems": ["<EMULATOR-GUID>", "<RECORDER-GUID>"]
      },
      "SubSystems": {
        "<EMULATOR-GUID>": {
// Chains/<RECORDER-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Outgoing
          "Outgoing": { "protocol": "TCP", "ip": "127.0.0.1", "port": 1234, "topics": ["Raw"] },
// Chains/<RECORDER-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Incoming
          "Incoming": {},
          "Controls": {
// Chains/<RECORDER-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Controls/<CONTROL-GUID-0>
            "<CONTROL-GUID-0>": { "type": "Slider", "label": "Clutter / Sector",
                                  "min": 0, "max": 3000, "value": "253" },
// Chains/<RECORDER-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Controls/<CONTROL-GUID-1>
            "<CONTROL-GUID-1>": { "type": "Slider", "label": "Tracks / Sector",
                                  "min": 0, "max": 10, "value": 10 }
          },
          "Data": {
            "Raw": {
// Chains/<RECORDER-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Data/Raw/Interpretation
              "Interpretation": { "key": "Raw", "type": "application/octet-stream",
                "dataTypes": "uint64,float,float,float,float",
                "header": "Time_ms,Range_m,Azimuth_deg,Speed_ms,Intensity"
              }
            }
          }
        }
      },
      "<RECORDER-GUID>": {
// Chains/<RECORDER-CHAIN-GUID>/SubSystems/<RECORDER-GUID>/Outgoing
        "Outgoing": { "protocol": "MQTT", "ip": "127.0.0.1", "port": 1883,
                      "topics": [] },
// Chains/<RECORDER-CHAIN-GUID>/SubSystems/<RECORDER-GUID>/Incoming
        "Incoming": { "protocol": "TCP", "ip": "127.0.0.1", "port": 1234,
```

```
                    "topics": ["Raw"], "source": "<EMULATOR-GUID>",
                    "layout":"uint64,float,float,float,float" },
       "Controls": {
// Chains/<RECORDER-CHAIN-GUID>/SubSystems/<RECORDER-GUID>/Controls/<CONTROL-GUID-0>
         "<CONTROL-GUID-0>": { "type": "TextBox", "label": "Output Directory", "value": "./" }}}}}}
```

## C.3.2 Data model - Non-persisted (data)

### C.3.2.1 Data data model

Listing C.9: Data model example (non-persisted)

```
{
  "AvailableSubSystems": {
    "<EMULATOR-GUID>": {
// AvailableSubSystems/<EMULATOR-GUID>/Status
      "Status": "Operational"
    },
    "<RECORDER-GUID>": {
// AvailableSubSystems/<RECORDER-GUID>/Status
      "Status": "Operational"
    }
  },
  "Chains": {
    "<RECORDER-CHAIN-GUID>": {
      "SubSystems": {
        "<EMULATOR-GUID>": {
// Chains/<RECORDER-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Rates
          "Rates": { "total": "555555", "errors": "000000" }
        },
        "<RECORDER-GUID>": {
// Chains/<RECORDER-CHAIN-GUID>/SubSystems/<RECORDER-GUID>/Rates
          "Rates": { "total": "555555", "errors": "000000" }}}}}}
```

# C.4 Playback chain

## C.4.1 Data model - Persisted (setup & control)

Listing C.10: Data model example (persisted)

```
{
// SelectedChain
  "SelectedChain": { "id": "<PLAYBACK-CHAIN-GUID>", "isRunning": true },
  "Chains": {
    "<PLAYBACK-CHAIN-GUID>": {
// Chains/<PLAYBACK-CHAIN-GUID>/Setup
      "Setup": {
        "label": "Playback",
        "origin": { "latitude": -33.9581, "longitude": 18.4601 },
        "range": "15000",
// Chains/<PLAYBACK-CHAIN-GUID>/Setup/SubSystems
        "SubSystems": ["<DATA-FEEDER-GUID>", "<PROCESSOR-GUID>"]
      },
      "SubSystems": {
        "<DATA-FEEDER-GUID>": {
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<DATA-FEEDER-GUID>/Outgoing
          "Outgoing": { "protocol": "MQTT", "ip": "127.0.0.1", "port": 1883, "topics": ["Raw"] },
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<DATA-FEEDER-GUID>/Incoming
          "Incoming": {},
          "Controls": {
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<DATA-FEEDER-GUID>/Controls/<CONTROL-GUID-0>
            "<CONTROL-GUID-0>": { "type": "Slider", "label": "Burst Interval [ms]",
             "min": 0, "max": 1000, "value": "250" },
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<DATA-FEEDER-GUID>/Controls/<CONTROL-GUID-1>
            "<CONTROL-GUID-1>": { "type": "TextBox", "label": "Input Directory", "value": "./" }
          },
          "Data": {
            "Raw": {
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<DATA-FEEDER-GUID>/Data/Raw/Interpretation
              "Interpretation": { "key": "Raw", "type": "application/octet-stream",
                "dataTypes": "uint64,float,float,float,float",
                "header": "Time_ms,Range_m,Azimuth_deg,Speed_ms,Intensity"
              }
            }
          }
        },
        "<PROCESSOR-GUID>": {
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Outgoing
          "Outgoing": { "protocol": "MQTT", "ip": "127.0.0.1", "port": 1883,
                        "topics": ["ClutterMap", "Plots"] },
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Incoming
          "Incoming": { "protocol": "MQTT", "ip": "127.0.0.1", "port": 1883, "topics": [null],
                        "source": "<DATA-FEEDER-GUID>" },
          "Controls": {
```

```
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Controls/<CONTROL-GUID-0>
            "<CONTROL-GUID-0>": { "type": "Slider", "label": "Plots Threshold",
                                  "min": 1, "max": 20, "value": "16" }
          },
          "Data": {
            "ClutterMap": {
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Data/ClutterMap/Interpretation
              "Interpretation": { "key": "ClutterMap", "type": "text/csv", "display": "HeatMap",
                "charset": "UTF-8", "dotSize": 100, "refreshPeriod": "PT4S",
                "dataTypes": "float,float,uint32",
                "header": "Latitude_deg,Longitude_deg,Intensity"
              }
            },
            "Plots": {
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Data/Plots/Interpretation
              "Interpretation": { "key": "Plots", "type": "text/csv", "display": "Plot",
                "charset": "UTF-8", "refreshPeriod": "PT4S",
                "dataTypes": "uint64,float,float,float,float,float,uint32",
                "header": "Time_ms,Latitude_deg,Longitude_deg,Range_m,Azimuth_rad,Speed_ms,Type",
                "classifications": [
                  { "label": "type0", "paletteIndex": 0 },
                  { "label": "type1", "paletteIndex": 1 },
                  { "label": "type2", "paletteIndex": 6 },
                  { "label": "type3", "paletteIndex": 12 }]}}}}}}}}
```

## C.4.2 Data model - Non-persisted (data)

### C.4.2.1 Data data model

Listing C.11: Data model example (non-persisted)

```
{
  "AvailableSubSystems": {
    "<DATA-FEEDER-GUID>": {
// AvailableSubSystems/<DATA-FEEDER-GUID>/Status
      "Status": "Operational"
    },
    "<PROCESSOR-GUID>": {
// AvailableSubSystems/<PROCESSOR-GUID>/Status
      "Status": "Operational"
    }
  },
  "Chains": {
    "<PLAYBACK-CHAIN-GUID>": {
      "SubSystems": {
        "<DATA-FEEDER-GUID>": {
// Chains/<PLAYBACK-CHAIN-GUID>/SubSystems/<DATA-FEEDER-GUID>/Rates
```

```
          "Rates": { "total": "555555", "errors": "000000" },
          "Data": {
            "Raw": {
// Chains/<PLAYBACK-CHAIN>/SubSystems/<DATA-FEEDER-GUID>/Data/Raw/Records
              "Records": /* RAW-RECORDS */
            }
          }
        },
        "<PROCESSOR-GUID>": {
// Chains/<PROCESSING-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Rates
          "Rates": { "total": "555555", "errors": "000000" },
          "Data": {
            "ClutterMap": {
// Chains/<PLAYBACK-CHAIN>/SubSystems/<PROCESSOR-GUID>/Data/ClutterMap/Records
              "Records": /* CLUTTER-RECORDS */
            },
            "Plots": {
// Chains/<PLAYBACK-CHAIN>/SubSystems/<PROCESSOR-GUID>/Data/Plots/Records
              "Records": /* PLOT-RECORDS */ }}}}}}}
```

#### C.4.2.2 Records

Listing C.12: "<DATA-FEEDER-GUID>/Data/Raw/Records" example

```
1630869348481,6035.0,125.5,50.0,4.0
1630869348481,10014.0,116.5,30.0,9.0
1630869348481,10805.0,119.5,14.0,5.0
```

Listing C.13: "<PROCESSOR-GUID>/Data/ClutterMap/Records" example

```
-33.89349099845401,18.435672922948648,6.0
-33.88899058371935,18.431567634595826,13.0
-33.91037590014317,18.4484546097588,13.0
```

Listing C.14: "<PROCESSOR-GUID>/Data/Plots/Records" example

```
1630868707430,-33.86011602940622,18.48721489970903,11154.0,13.0,97.33948516845703,2
1630868707430,-33.87208990906801,18.48390483024897,9791.0,13.0,85.44476318359375,1
1630868707430,-33.8840636764991,18.480593836263004,8428.0,13.0,73.55004119873047,2
```

## C.5 Processing chain

### C.5.1 Data model - Persisted (setup & control)

Listing C.15: Data model example (persisted)

```
{
// SelectedChain
  "SelectedChain": { "id": "<FULL-CHAIN-GUID>", "isRunning": true },
  "Chains": {
    "<FULL-CHAIN-GUID>": {
// Chains/<FULL-CHAIN-GUID>/Setup
      "Setup": {
        "label": "Primary chain",
        "origin": { "latitude": -33.9581, "longitude": 18.4601 },
        "range": "15000",
// Chains/<FULL-CHAIN-GUID>/Setup/SubSystems
        "SubSystems": ["<EMULATOR-GUID>", "<PROCESSOR-GUID>", "<TRACKER-GUID>"]
      },
      "SubSystems": {
        "<EMULATOR-GUID>": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Outgoing
          "Outgoing": { "protocol": "TCP", "ip": "127.0.0.1", "port": 1234, "topics": ["Raw"] },
// Chains/<FULL-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Incoming
          "Incoming": {},
          "Controls": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Controls/<CONTROL-GUID-0>
            "<CONTROL-GUID-0>": { "type": "Slider", "label": "Clutter / Sector",
                                  "min": 0, "max": 3000, "value": "171" },
// Chains/<FULL-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Controls/<CONTROL-GUID-1>
            "<CONTROL-GUID-1>": { "type": "Slider", "label": "Tracks / Sector",
                                  "min": 0, "max": 10, "value": "10" }
          },
          "Data": {
            "Raw": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Data/Raw/Interpretation
              "Interpretation": { "key": "Raw", "type": "application/octet-stream",
                "dataTypes": "uint64,float,float,float,float",
                "header": "Time_ms,Range_m,Azimuth_deg,Speed_ms,Intensity"
              }
            }
          }
        },
        "<PROCESSOR-GUID>": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Outgoing
          "Outgoing": { "protocol": "MQTT", "ip": "127.0.0.1", "port": 1883,
                        "topics": ["ClutterMap", "Plots"] },
// Chains/<FULL-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Incoming
          "Incoming": { "protocol": "TCP", "ip": "127.0.0.1", "port": 1234,
                        "topics": ["Raw"], "source": "<EMULATOR-GUID>",
```

```
                        "layout":"uint64,float,float,float,float" },
          "Controls": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Controls/<CONTROL-GUID-0>
            "<CONTROL-GUID-0>": { "type": "Slider", "label": "Plots Threshold",
                                  "min": 1, "max": 20, "value": "16" }
          },
          "Data": {
            "ClutterMap": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Data/ClutterMap/Interpretation
              "Interpretation": { "key": "ClutterMap", "type": "text/csv", "display": "HeatMap",
                "charset": "UTF-8", "dotSize": 100, "refreshPeriod": "PT4S",
                "dataTypes": "float,float,uint32",
                "header": "Latitude_deg,Longitude_deg,Intensity"
              }
            },
            "Plots": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Data/Plots/Interpretation
              "Interpretation": { "key": "Plots", "type": "text/csv", "display": "Plot",
                "charset": "UTF-8", "refreshPeriod": "PT4S",
                "dataTypes": "uint64,float,float,float,float,float,uint32",
                "header": "Time_ms,Latitude_deg,Longitude_deg,Range_m,Azimuth_rad,Speed_ms,Type",
                "classifications": [
                  { "label": "type0", "paletteIndex": 0 },
                  { "label": "type1", "paletteIndex": 1 },
                  { "label": "type2", "paletteIndex": 6 },
                  { "label": "type3", "paletteIndex": 12 }
                ]
              }
            }
          }
        },
        "<TRACKER-GUID>": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<TRACKER-GUID>/Outgoing
          "Outgoing": { "protocol": "MQTT", "ip": "127.0.0.1", "port": 1883,
                        "topics": ["Tracks"] },
// Chains/<FULL-CHAIN-GUID>/SubSystems/<TRACKER-GUID>/Incoming
          "Incoming": { "protocol": "MQTT", "ip": "127.0.0.1", "port": 1883,
                        "topics": [null], "source": "<PROCESSOR-GUID>" },
          "Controls": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<TRACKER-GUID>/Controls/<CONTROL-GUID-0>
            "<CONTROL-GUID-0>": { "type": "Slider", "label": "Required detections",
                                  "min": 2, "max": 6, "value": 4 }
          },
          "Data": {
            "Tracks": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<TRACKER-GUID>/Data/Tracks/Interpretation
              "Interpretation": { "key": "Tracks", "type": "text/csv", "display": "Track",
                "charset": "UTF-8", "trailLength": 500, "refreshPeriod": "PT4S",
                "dataTypes": "string_24,string_5,float,float,float,float,uint32,string_254",
                "header": "Time,Identifier,Latitude_deg,Longitude_deg,Bearing_deg,Speed_ms,Type,Info",
                "classifications": [
                  { "label": "type0", "symbolIndex": 0, "paletteIndex": 9 },
```

```
                { "label": "type1", "symbolIndex": 1, "paletteIndex": 11 },
                { "label": "type2", "symbolIndex": 2, "paletteIndex": 13 }]}}}}}}}}}
```

## C.5.2 Data model - Non-persisted (data)

### C.5.2.1 Data data model

Listing C.16: Data model example (non-persisted)

```
{
  "AvailableSubSystems": {
    "<EMULATOR-GUID>": {
// AvailableSubSystems/<EMULATOR-GUID>/Status
      "Status": "Operational"
    },
    "<PROCESSOR-GUID>": {
// AvailableSubSystems/<PROCESSOR-GUID>/Status
      "Status": "Operational"
    },
    "<TRACKER-GUID>": {
// AvailableSubSystems/<TRACKER-GUID>/Status
      "Status": "Operational"
    }
  },
  "Chains": {
    "<FULL-CHAIN-GUID>": {
      "SubSystems": {
        "<EMULATOR-GUID>": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<EMULATOR-GUID>/Rates
          "Rates": { "total": "555555", "errors": "000000" }
        },
        "<PROCESSOR-GUID>": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<PROCESSOR-GUID>/Rates
          "Rates": { "total": "555555", "errors": "000000" },
          "Data": {
            "ClutterMap": {
// Chains/<FULL-CHAIN>/SubSystems/<PROCESSOR-GUID>/Data/ClutterMap/Records
              "Records": /* CLUTTER-RECORDS */
            },
            "Plots": {
// Chains/<FULL-CHAIN>/SubSystems/<PROCESSOR-GUID>/Data/Plots/Records
              "Records": /* PLOT-RECORDS */
            }
          }
        },
        "<TRACKER-GUID>": {
// Chains/<FULL-CHAIN-GUID>/SubSystems/<TRACKER-GUID>/Rates
```

```
          "Rates": { "total": "245335", "errors": "000000" },
          "Data": {
            "Tracks": {
// Chains/<FULL-CHAIN>/SubSystems/<TRACKER-GUID>/Data/Tracks/Records
              "Records": /* TRACK-RECORDS */ }}}}}}
```

### C.5.2.2 Records

Listing C.17: "<PROCESSOR-GUID>/Data/ClutterMap/Records" example

```
-33.996316593900616,18.453037917333948,16.0
-33.97014626427901,18.451326270207176,17.0
-33.980650570041526,18.44367319883819,20.0
```

Listing C.18: "<PROCESSOR-GUID>/Data/Plots/Records" example

```
1631997191354,-33.99115438176117,18.43601824461468,4289.0,211.25,37.42953595456445,1
1631997191608,-33.96643140078091,18.446461634205562,1563.0,233.75,13.640094356955988,1
1631997191608,-33.973695480405816,18.43456624990128,2926.0,233.75,25.534815155760217,2
```

Listing C.19: "<TRACKER-GUID>/Data/Tracks/Records" example

```
"2021-09-18T20:33:11.609Z","00013",-33.96210861206055,18.491390228271484,177.0,25.534814834594727,1,"<pr...
"2021-09-18T20:33:11.613Z","00014",-33.96397399902344,18.505966186523438,181.0,37.429534912109375,1,"<pr...
"2021-09-18T20:33:11.617Z","00015",-33.96540832519531,18.474557876586914,191.0,13.640094757080078,1,"<pr...
```

# C.6 Spoofing chain

## C.6.1 Data model - Persisted (setup & control)

Listing C.20: Data model example (persisted)

```
{
// SelectedChain
  "SelectedChain": { "id": "<SPOOF-CHAIN-GUID>", "isRunning": true },
  "Chains": {
    "<SPOOF-CHAIN-GUID>": {
// Chains/<SPOOF-CHAIN-GUID>/Setup
      "Setup": {
        "label": "Spoofer",
        "origin": { "latitude": -33.9581, "longitude": 18.4601 },
        "range": "15000",
// Chains/<SPOOF-CHAIN-GUID>/Setup/SubSystems
        "SubSystems": ["<SPOOFER-GUID>"]
      },
      "SubSystems": {
        "<ADSB-GUID>": {
          "Controls": {
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Controls/<CONTROL-GUID-0>
            "<CONTROL-GUID-0>": { "type": "TextBox", "label": "Latitude Min", "value": "-45" },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Controls/<CONTROL-GUID-1>
            "<CONTROL-GUID-1>": { "type": "TextBox", "label": "Latitude Max", "value": "0" },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Controls/<CONTROL-GUID-2>
            "<CONTROL-GUID-2>": { "type": "TextBox", "label": "Longitude Min", "value": "0" },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Controls/<CONTROL-GUID-3>
            "<CONTROL-GUID-3>": { "type": "TextBox", "label": "Longitude Max", "value": "50" }
          },
          "Data": {
            "Tracks": {
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<ADSB-GUID>/Data/Tracks/Interpretation
              "Interpretation": { "key": "Tracks", "type": "text/csv", "display": "Track",
                "charset": "UTF-8", "refreshPeriod": "PT30S",
                "dataTypes": "string_24,string_5,float,float,float,float,uint32,string_254",
                "header": "Time,Identifier,Latitude_deg,Longitude_deg,Bearing_deg,Speed_ms,Type,Info",
                "classifications": [
                  { "label": "default", "symbolIndex": 0, "paletteIndex": 5 }
                ]
              }
            }
          }
        },
        "<SPOOFER-GUID>": {
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Outgoing
          "Outgoing": { "protocol": "MQTT", "ip": "127.0.0.1", "port": 1883,
                        "topics": ["ClutterMap", "Plots", "Strobes", "Tracks"] },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Incoming
```

```
          "Incoming": {},
          "Controls": {
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Controls/<CONTROL-GUID-0>
            "<CONTROL-GUID-0>": { "type": "Slider", "label": "Burst Interval [ms]",
                                  "min": 1, "max": 1000, "value": "250" },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Controls/<CONTROL-GUID-1>
            "<CONTROL-GUID-1>": { "type": "Radio","label": "Clutter send mode",
                                  "selected": "0", "items": ["Default", "Perf. Test"] },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Controls/<CONTROL-GUID-2>
            "<CONTROL-GUID-2>": { "type": "Slider", "label": "ClutterMap Rate [#/sector]",
              "min": 0, "max": 65536, "value": "50" },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Controls/<CONTROL-GUID-3>
            "<CONTROL-GUID-3>": { "type": "Radio", "label": "Plots send mode",
                                  "selected": "0", "items": ["Default", "Perf. Test" ] },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Controls/<CONTROL-GUID-4>
            "<CONTROL-GUID-4>": { "type": "Slider", "label": "Plots Rate [#/sector]",
                                  "min": 0, "max": 65536, "value": "8" },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Controls/<CONTROL-GUID-5>
            "<CONTROL-GUID-5>": { "type": "Slider", "label": "Strobes Rate [#/sector]",
                                  "min": 0, "max": 4096, "value": "1" },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Controls/<CONTROL-GUID-6>
            "<CONTROL-GUID-6>": { "type": "Slider", "label": "Tracks Rate (scanning) [#/sector]",
                                  "min": 0, "max": 512, "value": "1" },
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Controls/<CONTROL-GUID-7>
            "<CONTROL-GUID-7>": { "type": "Radio", "label": "Tracks send mode",
                                  "selected": "0", "items": ["Scanning (4Hz)", "Tracking (1Hz)"] }
          },
          "Data": {
            "ClutterMap": {
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Data/ClutterMap/Interpretation
              "Interpretation": { "key": "ClutterMap", "type": "text/csv", "display": "HeatMap",
                "charset": "UTF-8", "dotSize": 100, "refreshPeriod": "PT4S",
                "dataTypes": "float,float,uint32",
                "header": "Latitude_deg,Longitude_deg,Intensity"
              }
            },
            "Plots": {
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Data/Plots/Interpretation
              "Interpretation": { "key": "Plots", "type": "text/csv", "display": "Plot",
                "charset": "UTF-8", "refreshPeriod": "PT4S",
                "dataTypes": "uint64,float,float,float,float,float,uint32",
                "header": "Time_ms,Latitude_deg,Longitude_deg,Range_m,Azimuth_rad,Speed_ms,Type",
                "classifications": [
                  { "label": "type0", "paletteIndex": 0 },
                  { "label": "type1", "paletteIndex": 1 },
                  { "label": "type2", "paletteIndex": 6 },
                  { "label": "type3", "paletteIndex": 12 }
                ]
              }
            },
            "Strobes": {
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Data/Strobes/Interpretation
```

```
              "Interpretation": { "key": "Strobes", "type": "text/csv", "display": "Strobe",
                "charset": "UTF-8", "refreshPeriod": "PT4S",
                "dataTypes": "string,string,float,float,float,float,uint32",
                "header": "Time,Identifier,Origin_Latitude_deg,Origin_Longitude_deg,Range_m,Azimuth_deg,
                    Type",
                "classifications": [
                  { "label": "type0", "paletteIndex": 2 },
                  { "label": "type1", "paletteIndex": 8 },
                  { "label": "type2", "paletteIndex": 10 },
                  { "label": "type3", "paletteIndex": 13 }
                ]
              }
            },
            "Tracks": {
// Chains/<SPOOF-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Data/Tracks/Interpretation
              "Interpretation": { "key": "Tracks", "type": "text/csv", "display": "Track",
                "charset": "UTF-8", "trailLength": 3000, "refreshPeriod": "PT4S",
                "dataTypes": "string_24,string_5,float,float,float,float,uint32,string_254",
                "header": "Time,Identifier,Latitude_deg,Longitude_deg,Bearing_deg,Speed_ms,Type,Info",
                "classifications": [
                  { "label": "type0", "symbolIndex": 0, "paletteIndex": 2 },
                  { "label": "type1", "symbolIndex": 1, "paletteIndex": 4 },
                  { "label": "type2", "symbolIndex": 2, "paletteIndex": 13 }]}}}}}}}}
```

## C.6.2 Data model - Non-persisted (data)

### C.6.2.1 Data data model

Listing C.21: Data model example (non-persisted)

```
{
  "AvailableSubSystems": {
    "<SPOOFER-GUID>": {
// AvailableSubSystems/<SPOOFER-GUID>/Status
      "Status": "Operational"
    }
  },
  "Chains": {
    "<VIEWER-CHAIN-GUID>": {
      "SubSystems": {
        "<ADSB-GUID>": {
// Chains/<SPOOFER-CHAIN-GUID>/SubSystems/<SPOOFER-GUID>/Rates
          "Rates": { "total": "555555", "errors": "000000" },
          "Data": {
            "ClutterMap": {
// Chains/<SPOOFER-CHAIN>/SubSystems/<SPOOFER-GUID>/Data/ClutterMap/Records
              "Records": /* CLUTTER-RECORDS */
```

```
            },
            "Plots": {
// Chains/<SPOOFER-CHAIN>/SubSystems/<SPOOFER-GUID>/Data/Plots/Records
              "Records": /* PLOT-RECORDS */
            },
            "Strobes": {
// Chains/<SPOOFER-CHAIN>/SubSystems/<SPOOFER-GUID>/Data/Strobes/Records
              "Records": /* STROBE-RECORDS */
            },
            "Tracks": {
// Chains/<SPOOFER-CHAIN>/SubSystems/<SPOOFER-GUID>/Data/Tracks/Records
              "Records": /* TRACK-RECORDS */ }}}}}}
```

#### C.6.2.2 Records

Listing C.22: "<SPOOFER-GUID>/Data/ClutterMap/Records" example

```
-33.95373783055164,18.462883262470264,10
-33.9452889577018,18.468101833446177,10
-33.9448547121067,18.473672839992965,10
```

Listing C.23: "<SPOOFER-GUID>/Data/Plots/Records" example

```
1631993171097,-33.94452312454316,18.467014787163702,1636,23.0,181,1
1631993171097,-33.88687386759126,18.53833100326452,10712,42.5,69,1
1631993171097,-33.89587767710492,18.512337247648702,8424,35.0,223,3
```

Listing C.24: "<SPOOFER-GUID>/Data/Strobes/Records" example

```
"2021-09-18T19:26:10.860Z",8,-33.9581,18.4601,6393,28.5,2
"2021-09-18T19:26:10.860Z",7,-33.9581,18.4601,12283,39.5,0
"2021-09-18T19:26:10.860Z",6,-33.9581,18.4601,8393,43.0,2
```

Listing C.25: "<SPOOFER-GUID>/Data/Tracks/Records" example

```
"2021-09-18T19:26:10.861Z","  8",-33.88126024760922,18.538759794133544,130.5,149,0,"Extended Info:<br/>...
"2021-09-18T19:26:10.861Z","  7",-33.921122951128595,18.483682709900858,118.0,212,0,"Extended Info:<br/>...
"2021-09-18T19:26:10.860Z","  6",-33.91061627043476,18.484861895047,113.5,27,2,"Extended Info:<br/>This...
```

# Appendix D

# Subsystem Code Samples

This section contains several code snippets providing insight into usage of the Python-based assembly to implement various representative subsystems.

These are presented solely for demonstration purposes, as the general approach allows subsystem developers to modify the implementation as necessary. Modules requiring considerable interprocess communication (e.g. between Python and Matlab, as with recording and playback) are deemed out of scope here but may be considered as extensions of the code snippets presented here.

To this end, limited boilerplate code is provided as part of the *radar_subsystem* Python module, which will be made openly available. Its general sections are detailed here in the order of their definition, with customizable parts specifically indicated.

The modules referenced by the subsystem's logic must first be imported:

Listing D.1: Default imports

```
import asyncio
import os

import radar_subsystem
import yaml

# (1) [OPTIONAL] Subsystem specific imports

from radar_subsystem import Style
```

Asynchronous functions are widely used, with the *asyncio* Python module necessary for numerous core functions. The *os* and *yaml* Python modules are required to access and interpret the local configuration file (see Appendix B). The core classes, methods and functions for standard subsystem operation are contained in the *radar_subsystem* assembly.

This loop effectively implements the subsystem's Process component (see 5.1.4.3) and, as such, contains the subsystem's defining logic:

Listing D.2: General execution loop

```
async def loop_async(context, config):
    # (2) [OPTIONAL] Specify incoming/read queue
    # (3) [OPTIONAL] Specify outgoing/write queues
    # (4) [OPTIONAL] Specify control setup
    # (5) [OPTIONAL] Specify custom parameters
    # (6) Processing logic
```

The subsystem's read queue is assigned at (2) from the context and directly links to the Input Channel (where the subsystem realizes a Processor or Consumer). Several write queues may be assigned at (3) from the context and allow for direct use of the Output Channel (where the subsystem realizes a Generator or Processor). Where specific subsystem controls are defined for use from the HMI, these may be assigned at (4) from the context's controls array for ease of reference. Additional custom parameters may be assigned at (5), these will not be interpreted to the context and must be read directly from the config. The processing logic at (6) is wholly custom and its definition as required for the subsystem to perform its function.

The module's asynchronous main function serves as the subsystem entry point. This Python function will generally not require any specific updates:

Listing D.3: Entry point

```
async def main():
    context = None
    try:
        with open(os.path.join(os.path.dirname(__file__), 'config.yml'), 'r') as config_file:
            config = yaml.load(config_file, Loader=yaml.FullLoader)
    except:
        raise ValueError("Missing or invalid configuration provided.")
    # ------------------------------------------------------------------------
    context = radar_subsystem.Context(config)
    controller = radar_subsystem.Controller(context)
    # ------------------------------------------------------------------------
    try:
```

```
            await controller.start_async()
            await loop_async(context, config)
        except Exception as x:
            print(f'{Style.ERROR}process loop interrupted with:  -> \"{x}\"{Style.EOS}')
        finally:
            print(f"{Style.WARNING}terminating...{Style.EOS}")
            await controller.stop_async()
```

The function opens the local configuration file and then interprets it to an internal data context. The subsystem controller is then initialized from this data context. Once the controller is started up, the primary processing loop is executed.

Execution from the OS console or scripts is simplified by using the usual Python boilerplate statement:

Listing D.4: Script run

```
if __name__ == '__main__':
    # (8) [OPTIONAL] Initialize external processes
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        print(f"{Style.ERROR}process loop manually interrupted{Style.EOS}")
    except Exception as e:
        print(f"{Style.ERROR}{e}{Style.EOS}")
    finally:
        # (9) [OPTIONAL] Finalize external processes
        print('___\n')
        exit()
```

It ensures that the module's asynchronous functions are properly framed. Execution will terminate on specific interrupt by the user or on encountering an incident. External processes are also initialized (8) and terminated (9) here to ensure the availability of these processes for use by the primary process loop.

# D.1 Viewer code snippets

## D.1.1 Assignment

### D.1.1.1 Specify outgoing/write queues

Listing D.5: Specify outgoing/write queues

```
tracks_queue = context.output_channel.pipes['Tracks']['queue']
```

### D.1.1.2 Specify control setup

Listing D.6: Specify control setup

```
lat_min_textbox = context.controls[0]
lat_max_textbox = context.controls[1]
lng_min_textbox = context.controls[2]
lng_max_textbox = context.controls[3]
```

### D.1.1.3 Specify custom parameters

Listing D.7: Specify custom parameters

```
url = config['adsbUrl']
```

## D.1.2 Processing logic

Listing D.8: Processing logic

```
while not context.is_terminated:
    if not context.is_running:
        await asyncio.sleep(0.1)
        continue
    response = None
    try:
        response = requests.get(url.format(
            lat_min_textbox.value,
            lat_max_textbox.value,
            lng_min_textbox.value,
```

```
            lng_max_textbox.value))
    except Exception as x:
        print(f'{Style.ERROR}{x}{Style.EOS}')
    if (response is not None and response.ok):
        record = json.loads(response.content)
        track_states = record['states']
        if not track_states:
            continue
        for track_state in record['states']:
            # ---------------------------------------------------------
            if context.is_terminated:
                continue
            # ---------------------------------------------------------
            icao24, callsign, origin_country, _, _, longitude, latitude, baro_altitude, on_ground, velocity,
                true_track, vertical_rate, _, _, squawk, _, _ = track_state
            velocity = velocity if velocity else 0
            vertical_rate = vertical_rate if vertical_rate else 0
            baro_altitude = baro_altitude if baro_altitude else 0
            climb_symbol = "&#8230;" if on_ground else "&#x25B2;" if vertical_rate > 0 else
                "&#x25BC;" if vertical_rate < 0 else "&#x25C7;"
            climb_colour = "goldenrod" if on_ground else "lime" if vertical_rate > 0 else
                "tomato" if vertical_rate < 0 else "grey"
            # ---------------------------------------------------------
            tracks_queue.put([
                radar_subsystem.timestamp(),
                icao24,
                float(latitude),
                float(longitude),
                float(true_track),
                float(velocity),
                0,
                f"<b>{callsign}</b> ({squawk})<br/>" +
                f"<em>{origin_country}</em><hr/>" +
                f"ALT: {int(baro_altitude)} m<br/>" +
                f"C/R: <p style='display:inline;color:{climb_colour};'><b>{climb_symbol}</b></p>" +
                    f"{abs(float(vertical_rate))} m/s<br/>" +
                "</p>"
            ])
    await asyncio.sleep(10)
```

## D.2 Control code snippets

### D.2.1 Assignment

#### D.2.1.1 Specify control setup

Listing D.9: Specify control setup

```
print_to_console_radiogroup = context.controls[0]
write_mechanism_radiogroup = context.controls[1]
control_textbox = context.controls[2]
control_togglegroup = context.controls[3]
control_radiogroup = context.controls[4]
control_slider = context.controls[5]
```

#### D.2.1.2 Specify custom parameters

Listing D.10: Specify custom parameters

```
with open(os.path.join(os.path.dirname(__file__), control_file_name + '.dat'), "wb") as f:
    f.truncate(16384)
```

### D.2.2 Processing logic

Listing D.11: Processing logic

```
with open(os.path.join(os.path.dirname(__file__), control_file_name + '.dat'), "r+b") as f:
    mm = mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_WRITE)
    buffer = memoryview(mm)
    # ------------------------------------------------------------------
    ex_textbox.set_map_range(0)
    ex_togglegroup.set_map_range(ex_textbox.end_pos)
    ex_radiogroup.set_map_range(ex_togglegroup.end_pos)
    ex_slider.set_map_range(ex_radiogroup.end_pos)
    # ------------------------------------------------------------------
    ex_textbox.observe('received', lambda: ex_textbox.write_to_mem_map(
        buffer) if write_mechanism_radiogroup.selected == 0 else None)
    ex_togglegroup.observe('received', lambda: ex_togglegroup.write_to_mem_map(
        buffer) if write_mechanism_radiogroup.selected == 0 else None)
    ex_radiogroup.observe('received', lambda: ex_radiogroup.write_to_mem_map(
        buffer) if write_mechanism_radiogroup.selected == 0 else None)
    ex_slider.observe('received', lambda: ex_slider.write_to_mem_map(
```

```
        buffer) if write_mechanism_radiogroup.selected == 0 else None)
    # ---------------------------------------------------------------------
    while not context.is_terminated:
        if context.is_running and write_mechanism_radiogroup.selected == 1:
            ex_textbox.write_to_mem_map(buffer)
            ex_togglegroup.write_to_mem_map(buffer)
            ex_radiogroup.write_to_mem_map(buffer)
            ex_slider.write_to_mem_map(buffer)
        # -----------------------------------------------------------------
        if context.is_running and print_to_console_radiogroup.selected == 0:
            print(f"""---
{datetime.datetime.utcnow()}
TextBox value:  {ex_textbox.value}
CheckBox items: {ex_togglegroup.items[0].label}:{ex_togglegroup.items[0].is_checked}
            {ex_togglegroup.items[1].label}:{ex_togglegroup.items[1].is_checked}
Radio selected: {ex_radiogroup.items[ex_radiogroup.selected]}
Slider value:   {ex_slider.value}""")
        # -----------------------------------------------------------------
        await asyncio.sleep(1)
```

## D.3 Emulator code snippets

### D.3.1 Assignment

#### D.3.1.1 Specify outgoing/write queues

Listing D.12: Specify outgoing/write queues

```
raw_queue = context.output_channel.pipes['Raw']['queue']
```

#### D.3.1.2 Specify control setup

Listing D.13: Specify control setup

```
clutter_slider = context.controls[0]
tracks_slider = context.controls[1]
```

#### D.3.1.3 Specify custom parameters

Listing D.14: Specify custom parameters

```
block_sequence = 0
comp_start_time_msec = round(time.time_ns() * 1E-6)
plot_radials = []
for i in range(0, blocks_count):
    plot_radials.append([])
    for j in range(0, max_tracks_per_block):
        plot_radials[i].append(
            [(j + 1) * int(max_range / (max_tracks_per_block + 1)), (i + 0.5) * block_azimuth_span])
```

### D.3.2 Processing logic

Listing D.15: Processing logic

```
while not context.is_terminated:
    if not context.is_running:
        await asyncio.sleep(0.1)
        continue
    # ----------------------------------------------------------------------
```

```
    start_angle = block_sequence * block_azimuth_span
    comp_start_time_msec = round(time.time_ns() * 1E-6)
    for _ in range(0, int(clutter_slider.value)):
        # ---------------------------------------------------------------
        if context.is_terminated:
            continue
        # ---------------------------------------------------------------
        rng = random.randint(0, max_range)
        az = start_angle + \
            (random.randint(0, randomization_factor) * azimuth_resolution)
        intensity = random.randint(0, cutoff_clutter_intensity)
        speed = random.randint(0, cutoff_clutter_speed_ms)
        # ---------------------------------------------------------------
        raw_queue.put([
            comp_start_time_msec,
            rng,
            az,
            speed,
            intensity
        ])
    for j in range(0, int(tracks_slider.value)):
        # ---------------------------------------------------------------
        if context.is_terminated:
            continue
        # ---------------------------------------------------------------
        plot_radials[block_sequence][j][0] = plot_radials[block_sequence][j][0] + \
            range_increment if plot_radials[block_sequence][j][0] < max_range else min_range
        plot_radials[block_sequence][j][1] = plot_radials[block_sequence][j][1] + \
            azimuth_increment if plot_radials[block_sequence][j][1] < 360 else azimuth_increment
        intensity = random.randint(
            cutoff_clutter_intensity + 1, max_clutter_intensity)
        speed = plot_radials[block_sequence][j][0] * tan_05
        # ---------------------------------------------------------------
        raw_queue.put([
            comp_start_time_msec,
            plot_radials[block_sequence][j][0],
            plot_radials[block_sequence][j][1],
            speed,
            intensity
        ])
    comp_delta_time_msec = round(
        time.time_ns() * 1E-6) - comp_start_time_msec
    block_sequence = (block_sequence + 1) % blocks_count
    if comp_delta_time_msec < block_time_msec:
        await asyncio.sleep((block_time_msec - comp_delta_time_msec) * 1E-3)
```

## D.4 Processing code snippets

### D.4.1 Assignment

#### D.4.1.1 Specify incoming/read queue

Listing D.16: Specify incoming/read queue

```
read_queue = context.input_channel.queue
```

#### D.4.1.2 Specify outgoing/write queues

Listing D.17: Specify outgoing/write queues

```
clutter_write_queue = context.output_channel.pipes['ClutterMap']['queue']
plots_write_queue = context.output_channel.pipes['Plots']['queue']
```

#### D.4.1.3 Specify control setup

Listing D.18: Specify control setup

```
threshold_slider = context.controls[0]
```

#### D.4.1.4 Specify custom parameters

Listing D.19: Specify custom parameters

```
destination = [0, 0]
intensity = 0
intensity_value = 0
range_value = 0
azimuth_value = 0
```

### D.4.2 Processing logic

Listing D.20: Processing logic

```
while not context.is_terminated:
    if not context.is_running or not read_queue or read_queue.empty():
        await asyncio.sleep(0.1)
        continue
    while context.is_running and clutter_write_queue and read_queue and not read_queue.empty():
        for time_ms, range, azimuth, speed, intensity in context.input_channel.unpack():
            range_value = float(range)
            azimuth_value = float(azimuth)
            destination = distance(meters=range_value).destination(
                context.sensor_origin, azimuth_value)
            intensity_value = float(intensity)
            clutter_write_queue.put(
                [destination.latitude, destination.longitude, intensity_value])
            if intensity_value >= threshold_slider.value:
                plots_write_queue.put([int(time_ms), destination.latitude, destination.longitude,
                    range_value, azimuth_value, float(speed),
                        1 if (intensity_value >= 18) else 2 if (intensity_value >= 16) else 3])
```

## D.5 Spoofing code snippets

### D.5.1 Assignment

#### D.5.1.1 Specify outgoing/write queues

Listing D.21: Specify outgoing/write queues

```
cluttermap_queue = context.output_channel.pipes['ClutterMap']['queue']
plots_queue = context.output_channel.pipes['Plots']['queue']
strobes_queue = context.output_channel.pipes['Strobes']['queue']
tracks_queue = context.output_channel.pipes['Tracks']['queue']
```

#### D.5.1.2 Specify control setup

Listing D.22: Specify control setup

```
interval_slider = context.controls[0]
cluttermap_mode = context.controls[1]
cluttermap_slider = context.controls[2]
plots_mode = context.controls[3]
plots_slider = context.controls[4]
strobes_slider = context.controls[5]
tracks_mode = context.controls[6]
tracks_slider = context.controls[7]
```

#### D.5.1.3 Specify custom parameters

Listing D.23: Specify custom parameters

```
_cluttermap_buffer = []
_plots_buffer = []
_strobes_buffer = []
_tracks_buffer = []
block_send_time_msec = 0
block_sequence = 0
block_start_angle = 0
sector_count = 0
```

### D.5.2 Processing logic

Listing D.24: Processing logic

```
while not context.is_terminated:
    if not context.is_running:
        await asyncio.sleep(0.1)
        continue
    # -----------------------------------------------------------------
    current_time_msec = int(round(time.time_ns() * 1E-6))
    if current_time_msec > block_send_time_msec:
        block_send_time_msec = current_time_msec
    await asyncio.sleep(max(block_send_time_msec - current_time_msec, 1) * 1E-03)
    # -----------------------------------------------------------------
    if cluttermap_slider.value > 0:
        while len(_cluttermap_buffer) > 0:
            cluttermap_queue.put(_cluttermap_buffer.pop())
    else:
        _cluttermap_buffer.clear()
    if context.is_running and plots_slider.value > 0:
        while len(_plots_buffer) > 0:
            plots_queue.put(_plots_buffer.pop())
    else:
        _plots_buffer.clear()
    if context.is_running and strobes_slider.value > 0:
        while len(_strobes_buffer) > 0:
            strobes_queue.put(_strobes_buffer.pop())
    else:
        _strobes_buffer.clear()
    if context.is_running and tracks_slider.value > 0:
        if tracks_mode.selected == 0:
            while len(_tracks_buffer) > 0:
                tracks_queue.put(_tracks_buffer.pop())
        elif sector_count % 16 == 0:
            while len(_tracks_buffer) > 0:
                tracks_queue.put(_tracks_buffer.pop())
    else:
        _tracks_buffer.clear()
    # -----------------------------------------------------------------
    block_send_time_msec = block_send_time_msec + int(interval_slider.value)
    block_start_angle = block_sequence * 22.5
    block_sequence = (block_sequence + 1) % 16
    # -----------------------------------------------------------------
    if cluttermap_slider.value > 0:
        for i in range(0, int(cluttermap_slider.value)):
            # -------------------------------------------------------------
            if context.is_terminated:
                continue
            # -------------------------------------------------------------
            if cluttermap_mode.selected == 0:
                rng = random.randint(0, 15000)
                az = block_start_angle + (random.randint(0, 44) * 0.5)
```

```
            destination = distance(meters=rng).destination(
                context.sensor_origin, az)
            intensity = 10;#random.randint(1, 10)
        else:
            destination = context.sensor_origin
            intensity = 10
        # -----------------------------------------------------------
        _cluttermap_buffer.append([
            destination.latitude,
            destination.longitude,
            intensity
        ])
if plots_slider.value > 0:
    for i in range(0, int(plots_slider.value)):
        # -----------------------------------------------------------
        if context.is_terminated:
            continue
        # -----------------------------------------------------------
        if plots_mode.selected == 0:
            rng = random.randint(0, 15000)
            az = block_start_angle + (random.randint(0, 44) * 0.5)
            destination = distance(meters=rng).destination(
                context.sensor_origin, az)
            speed = random.randint(1, 300)
            type = random.randint(0, 3)
        else:
            rng = 15000
            az = 100
            destination = context.sensor_origin
            speed = 300
            type = 0
        # -----------------------------------------------------------
        _plots_buffer.append([
            block_send_time_msec,
            destination.latitude,
            destination.longitude,
            rng,
            az,
            speed,
            type
        ])
if strobes_slider.value > 0:
    for i in range(0, int(strobes_slider.value)):
        # -----------------------------------------------------------
        if context.is_terminated:
            continue
        # -----------------------------------------------------------
        rng = random.randint(0, 15000)
        az = block_start_angle + (random.randint(0, 44) * 0.5)
        type = random.randint(0, 3)
        # -----------------------------------------------------------
        _strobes_buffer.append([
```

```
            radar_subsystem.timestamp(),
            (block_sequence * int(strobes_slider.value)) + i,
            context.sensor_origin.latitude,
            context.sensor_origin.longitude,
            rng,
            az,
            type
        ])
if tracks_slider.value > 0:
    for i in range(0, int(tracks_slider.value)):
        # -------------------------------------------------------------
        if context.is_terminated:
            continue
        # -------------------------------------------------------------
        rng = random.randint(0, 15000)
        az = block_start_angle + (random.randint(0, 44) * 0.5)
        destination = distance(meters=rng).destination(
            context.sensor_origin, az)
        bearing = az + 90
        speed = random.randint(1, 300)
        type = random.randint(0, 2)
        # -------------------------------------------------------------
        _tracks_buffer.append([
            radar_subsystem.timestamp(),
            f"{((block_sequence * int(tracks_slider.value)) + i):5}",
            destination.latitude,
            destination.longitude,
            bearing,
            speed,
            type,
            "Extended Info:<br/>This is completely dynamic</br> and can place " +
                "<p style='color:red;'><b>whatever</b></p>you want here.</p>"
        ])
sector_count = sector_count + 1
```