

A fault-tolerant wireless distributed computing platform for robot swarms

PJ Rossouw

 [orcid.org 0000-0001-7162-6324](https://orcid.org/0000-0001-7162-6324)

Dissertation accepted in partial fulfilment of the requirements for the degree *Master of Science in Computer Science* at the North-West University

Supervisor:	Mr H Foulds
Co-supervisor:	Prof GR Drevin
Assistant Supervisor:	Prof L Drevin

Graduation May 2020

23598840

Acknowledgements

Firstly, I will thank Marinda who has been immensely supportive throughout this research. This research has been a part of our lives since you were my girlfriend, fiancée and now my wife. Thank you for your love, patience and effort.

To my parents, JP and Margaret, thank you for setting me up for success in life. To my in-laws and friends, thanks for understanding the sacrifices and celebrating the little victories with me.

I want to express my appreciation for my supervisors Henry Foulds, Gunther Drevin and Lynette Drevin for their guidance in developing me as a researcher. To Luke Machowski, thank you for the insightful conversations - I'm glad that they won't end with this research.

To Isabel Swart, thank you for the language editing of this research and making sure it presents as well as it should.

Finally, I will thank God for the opportunities he has given me, the people He has put on my path and for my desire to learn.

Abstract

A novel use of drone swarms was demonstrated by Intel during the opening ceremony of the 2018 Winter Olympics when hundreds of LED-equipped drones flew in precise formation to create a visual spectacle. This show was controlled and coordinated from a central control system that was responsible for all processing. From a systems architecture perspective, this system represents a single point of failure in their system. In order to determine whether a decentralised implementation could replace the centralised control hub, a process of iterative systems architecture is conducted. The aim of the research is to produce a fault-tolerant wireless distributed computing platform. The creation of this artefact is guided by the design and creation methodology. The final iteration of the artefact enhances fault-tolerance by leveraging container orchestration, microservice architecture, event-streaming and an IEEE 802.11s wireless mesh network. The artefact was constructed using affordable commodity hardware and open-source software and demonstrated fault-tolerance in several scenarios while facilitating wireless distributed computing.

Key terms: DISTRIBUTED COMPUTING, FAULT-TOLERANCE, ROBOT SWARMS, SYSTEMS ARCHITECTURE, WIRELESS MESH NETWORKS

Opsomming

'n Nuwe gebruik van hommeltuig-swerms is deur Intel gedemonstreer by die openingseromonie van die 2018 Winter Olimpiese spele toe honderde hommeltuie met LED ligte in presiese formasie gevlieg het om 'n visuele skouspel te skep. Hierdie vertoning is beheer en gekoördineer vanaf 'n sentrale beheersentrum wat verantwoordelik was vir alle verwerking. Vanuit 'n stelselargitektuur perspektief verteenwoordig hierdie stelsel 'n enkele punt van mislukking. Ten einde te bepaal of 'n gedesentraliseerde implementasie die gesentraliseerde beheersentrum kan vervang, word 'n proses van iteratiewe stelselargitektuur uitgevoer. Die doel van die navorsing is om 'n foutverdraagsame draadloos verspreide verwerking platform te ontwikkel, en die skepping van hierdie artefak word gelei deur die ontwerp- en skeppingsmetodologie. Die finale iterasie van die artefak bevorder fouttoleransie deur gebruik te maak van houerorkestrasie, mikrosdiens-argitektuur, gebeurtenis-stroom en 'n IEEE 802.11s draadlose maas netwerk. Die artefak is gebou met behulp van bekostigbare kommoditeitshardeware en oop-bron sagteware en het foutverdraagsaamheid in verskeie omstandighede gedemonstreer terwyl dit draadlose verspreide verwerking fasiliteer.

Sleuteltermes: DRAADLOSE MAAS NETWERKE, FOUTTOLERANSIE, ROBOT SWERMS, STELSELS ARGITEKTUUR, VERSPREIDE VERWERKING

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation was sent for professional language editing in accordance with the University's requirements and the certificate of confirmation follows this declaration.

This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified above, in the text, and acknowledgements.

Pieter Rossouw
November 2019

This serves to confirm that I, Isabella Johanna Swart, registered with and accredited as professional translator by the South African Translators' Institute, registration number 1001128, language edited Chapters 1-5 and Appendix 1 of the following dissertation:

A fault-tolerant wireless distributed computing platform for robot swarms

by

PJ Rossouw

A handwritten signature in black ink, appearing to read 'ISwart', with a stylized, flowing script.

Dr Isabel J Swart

Date: 14 November 2019

23 Poinsettia Close
Van der Stel Park
Dormehlsdrift
GEORGE
6529

Tel: (044) 873 0111

Cell: 082 718 4210

e-mail: isaswart@telkomsa.net

Contents

List of Figures	xv
List of Scripts	xix
List of Tables	xxiii
1 Introduction	3
1.1 Contextualisation	5
1.1.1 Robustness	5
1.1.2 Wireless networking	5
1.1.3 Distributed computing	7
1.2 Research and application areas	8
1.3 Problem statement	9
1.4 Aim and objectives	9
1.5 Research paradigm	10
1.6 Research approach	10
1.6.1 Steps and guidelines	10
1.6.2 Research outputs	12
1.6.3 Evaluation methods	12
1.7 Summary	13
2 Literature Review	15
2.1 Platform robustness	15
2.1.1 System faults	17
2.1.2 Fault-detection mechanisms	18
2.1.3 Fault-tolerance approaches	19
2.1.3.1 Proactive fault-tolerance approaches	19
2.1.3.2 Reactive fault-tolerance approaches	20
2.2 Wireless Networking	21
2.2.1 OSI reference network model	21

2.2.2	IEEE 802.11 wireless networks	22
2.2.3	Challenge 1: Crowded 2.4 GHz spectrum	23
2.2.4	Challenge 2: High packet loss rate and ossified transport protocols	26
2.2.5	Network performance evaluation	28
2.3	Distributed Computing	30
2.3.1	Classifying distributed systems	31
2.3.2	Distributed and parallel computing	32
2.3.2.1	Clusters	35
2.3.2.2	Grids	36
2.3.2.3	Cloud computing	37
2.3.3	Wireless distributed computing	37
2.4	System Architecture	39
2.4.1	Monoliths	39
2.4.2	Microservices	40
2.4.3	Event-driven architecture	42
2.4.4	Containerisation	45
2.5	Swarm robotics	47
2.5.1	Swarm robustness	48
2.5.2	Swarm flexibility	48
2.5.3	Swarm scalability	49
2.5.4	RoboCup	49
2.5.4.1	Robot networking	49
2.5.4.2	Robot processing	50
2.6	Summary	51
3	Artefact Creation	53
3.1	Illustrative scenario	53
3.2	Platform hardware	54
3.3	Platform networking	55
3.4	Platform software iterations	57

3.4.1	Iteration 1: MPICH	58
3.4.2	Iteration 2: WebSockets	61
3.4.3	Iteration 3: RPC	63
3.4.4	Iteration 4: Streaming implementations	69
3.4.4.1	Apache Kafka	69
3.4.4.2	NATS Streaming	72
3.5	Containerisation of the platform	77
3.6	Summary	80
4	Artefact Evaluation	83
4.1	Artefact evaluation using technical experiments	84
4.1.1	Network experiments	84
4.1.1.1	Experiment 1: The effect of transmission distance on network performance	84
4.1.1.2	Experiment 2: Comparison between infrastructure and mesh networks	87
4.1.1.3	Experiment 3: Measuring the impact of the Docker overlay network	91
4.1.2	Event-streaming experiments	96
4.1.2.1	Experiment 4: Message size	97
4.1.2.2	Experiment 5: Number of publishers and subscribers	98
4.1.2.3	Experiment 6: Event persistence	99
4.2	Summary	101
5	Conclusion	103
5.1	Result summary	104
5.2	Key findings	105
5.2.1	Fault-tolerance enhancements of the WDC platform	106
5.2.2	Performance evaluation of the WDC platform	107
5.3	Future work	109
5.3.1	Investigate the QoS metrics of multi-hop transmissions	109
5.3.2	Revisit Apache Kafka with a more powerful SBC	109

5.3.3	Develop a network performance analysis method for use in environments with packet encapsulation	109
5.3.4	Compare container orchestrator solutions for low-powered WDC	110
5.3.5	Investigate the use of alternative networking standard to improve swarm range	111
5.4	Personal reflection	111
Bibliography		113
Appendix A Additional figures and tables		123
A.1	Wireless Networking	123
A.2	Distributed computing	124
A.3	System architecture	127
Appendix B Artefact Setup		129
B.1	Platform configuration	129
B.1.1	Hardware choices	129
B.1.2	OS setup and configuration	129
B.1.3	Mesh networking setup	131
B.1.4	Docker installation and Swarm configuration	133
B.2	Application configuration	134
B.2.1	Kafka configuration	134
B.2.2	Kafka stream operator development in Node.js	135
B.2.3	NATS configuration	136
B.2.4	Application deployment	136

List of Figures

1.1	Research map extract for Chapter 1. The complete research map is presented on page 1.	3
1.2	<i>Left:</i> Intel drone swarm at the 2018 Winter Olympics opening ceremony [5]. <i>Right:</i> Intel drone team controlling a similar demonstration from a centralised control centre [6].	4
1.3	The theoretical fields supporting this research.	9
1.4	The engineering cycle [36]. Question marks indicate knowledge questions and exclamation marks indicate design problems. The first iteration of the cycle starts at <i>Problem investigation</i>	11
2.1	Research map extract for Chapter 2. The complete research map is presented on page 1.	15
2.2	Categorisation of faults in cloud computing environments [40].	17
2.3	Redundant power supply unit (PSU) for servers/workstations [41] that represents a proactive mitigation against a physical layer fault.	18
2.4	Fault-tolerance approaches in cloud computing environments [40].	19
2.5	Representation of Wi-Fi channels with 802.11g/n 20 MHz channels and 40 MHz (802.11n only) “wide channels”. Figure adapted from representations by [57], [58]. . .	24
2.6	Channels available in the Unlicensed National Information Infrastructure (U-NII) bands within the 5 GHz spectrum. Channel bonding is applied in IEEE 802.11n and 802.11ac to simultaneously use multiple standard 20 MHz wide channels to support increased throughput. Figure adapted from work by [58].	25
2.7	Snoop vs. Regular TCP comparison: sequence numbers of received frames against time [60].	27
2.8	MIMD architecture using shared memory [70].	31
2.9	MIMD architecture using message passing [70].	31
2.10	Workflow from the distributed robotics domain [76]. The implementation is an example of a workflow with a mix of task parallelism and data parallelism.	34
2.11	Relationship between distributed computing solutions [77].	35
2.12	MPI application architecture. Figure adapted from [81].	36
2.13	Swarm of drones in which each drone has local sensing, computation and networking capabilities.	38

2.14	Using stream processor microservices to build organisational processes as the flow of atomic events [95].	42
2.15	Comparison of request-driven architecture to event-driven streaming architecture [73].	43
2.16	Evolutionary perspective of system architectures [98].	44
2.17	Potential future of system architectures [98].	45
2.18	Comparison between Multi-core Processing, Virtualisation and Containerisation [100]. <i>CR</i> : Processing core, <i>OS</i> : Operating System, <i>SW</i> : Software process, <i>VM</i> : Virtual Machine, <i>CN</i> : Container.	46
2.19	Fully autonomous soccer match at RoboCup 2017 [112].	50
2.20	Network topologies that have to be supported by a RANET protocol [113].	51
3.1	Research map extract for Chapter 3. The complete research map is presented on page 1.	53
3.2	The combination of computation and networking hardware used to construct the platform. Note that for the experiments presented in Chapter 4, the units were separated from this rack to ensure a specific distance between network adapters. This was necessary due to the short length of the replacement cables.	55
3.3	Comparison between mesh routing protocols. Figure redesigned from the work of Pojda et al. [120].	57
3.4	High-level architecture for the MPICH implementation.	58
3.5	High-level architecture for the WebSocket implementation.	62
3.6	Software components of the WebSocket drone swarm application.	63
3.7	Extract from drone swarm application dashboard: 3D position plot with trails. . . .	64
3.8	Extract from drone swarm application dashboard: distance between nodes and average speed.	64
3.9	High-level architecture for the Jayson RPC implementation. Compute servers are marked <i>S</i> while clients are marked <i>C</i> . The arrows represent function calls, with each client being able to call functions from any server with load balancing.	65
3.10	Architecture for the RPC implementation. Multiple decoupled compute servers provide service to independent clients. Two compute servers are presented in this figure, but the system can run with one, two or more compute servers.	66
3.11	Asynchronous execution: states of a JavaScript Promise [127].	67
3.12	Three instances of the RPC jayson compute server, reporting the number of requests served per second.	67

3.13	Three instances of the compute client instance, each printing which method it is requesting and any correctional action it is taking. Nodes are identified with their unique IDs and each of the indicated areas shows an example of a corrective action being taken.	68
3.14	Client response when a compute server becomes unavailable. Note that after the error (marked red), the next request is fulfilled (marked blue) by another compute node. The servers are running locally in this example.	68
3.15	High-level architecture for the Apache Kafka implementation.	71
3.16	High-level architecture for the WebSocket implementation.	73
3.17	NATS Streams server dedicated to the drone swarm simulation.	74
3.18	NATS Streams implementation of the drone swarm simulation.	75
3.19	Streaming architecture implementation of the drone swarm simulation.	76
3.20	Comparison of interest between Kubernetes and Docker Swarm based on Google Trends data.	79
3.21	Architecture diagram of the fault-tolerant wireless distributed computing platform. .	82
4.1	Research map extract for Chapter 4. The complete research map is presented on page 1.	83
4.2	Representation of the distance experiment setup. The default ping parameters were used; a standard 56 byte ICMP packet once per second for 30 seconds.	85
4.3	Distance profiles of the four Quality of Service metrics.	86
4.4	Near-field, transitional and far-field regions in wireless communications (Figure adapted from [140]). The distance of one wavelength in 2.4 GHz transmissions, as is typically used in IEEE 802.11n, is approximately 12.5 cm. The reactive zone within the near-field region is indicated at 0.159α	87
4.5	Infrastructure vs. mesh-point mode configurations. The design of the platform (presented in Figure 3.21 on page 82) favours the mesh-point network mode for increased fault-tolerance. Note that while this figure presents four nodes, only three nodes were used in the platform.	88
4.6	Mean bandwidth comparison over 30 seconds between infrastructure and mesh network modes.	89
4.7	Mean latency comparison over 30 seconds between infrastructure and mesh network modes.	89

4.8	Mean jitter comparison over 30 seconds between infrastructure and mesh network modes.	90
4.9	Packet loss comparison between infrastructure and mesh network modes.	90
4.10	Comparison of native host networks versus Docker Swarm overlay network.	91
4.11	Example output of the <i>wavemon</i> utility used for displaying live wireless networking information when connected in a mesh. In this example, it is presenting the information on the <i>raspi-mesh</i> SSID, but it was also used to measure signal quality in tests using infrastructure wireless modes.	92
4.12	Bandwidth comparison between direct communication over 802.11n network and the Docker Swarm overlay network.	93
4.13	Latency comparison between direct networking over 802.11n network and Docker Swarm overlay network.	93
4.14	Packet loss % comparison between direct networking over 802.11n network and Docker Swarm overlay network.	94
4.15	Jitter comparison between direct networking over 802.11n network and Docker Swarm overlay network.	95
4.16	Relationship between message size and throughput. The y-axis on the left presents the network utilisation with blue bars indicating the values. The y-axis on the right presents the message throughput and is indicated by the red line.	97
4.17	Comparison of message rates between pairs of publishers and subscribers linked to the same topic on the same node. This test was run at a fixed message size of 1024 bytes and results are grouped by the number of publishers (1P,2P and 3P).	98
4.18	Corresponding configuration files and NATS Streaming startup output.	99
4.19	Comparison of message rates between memory-persistence and file-persistence modes of the NATS Streaming Server.	101
5.1	Research map extract for Chapter 5. The complete research map is presented on page 1.	103
5.2	Comparison of processor performance in millions of instructions per second (MIPS) between generations of Raspberry Pi models. Figure created using data from original work by Zwetsloot [144].	110
A.1	A Wi-Fi analyser mobile app can visualise Wi-Fi channels and recommend the “best” channels to use. The app visualises the crowded nature of wireless channels in highly populated areas.	123

A.2	Identified requirements for an evolutionary transport-layer framework as a solution to the ossified transport protocols in use today. Figure adapted from [62].	124
A.3	Workflow from the healthcare domain [76]. The implementation is an example of a purely task parallel workflow.	126
A.4	Workflow from the computer vision domain [76]. The implementation is an example of a purely data parallel workflow.	126
A.5	Architecture overview of the FX Core system used at a major bank in Denmark, implemented using a legacy monolith architecture [93].	127
A.6	Architecture overview of the FX Core system used at a major bank in Denmark, implemented using a microservice architecture [93].	128
B.1	Improved cooling of Raspberry Pi's using an open airflow case and aluminium heatsinks.	130
B.2	Functional Docker swarm on a mesh network.	134

List of Scripts

3.1	Result of deploying the Docker Stack to provision the application.	80
4.1	Extract from nats-bench automation script. The command executes the nats-bench utility with parameters, ready to process and aggregate the results.	96
B.1	Dependency installation on DietPi. Some dependencies are already included in Raspbian, but have to be manually installed when using the light DietPi image. . . .	130
B.2	Raspberry Pi BATMAN installation.	131
B.3	Bash script that initialises the mesh network on each node.	132
B.4	Entry in the <i>crontab</i> that runs Script B.3 at every boot. This allows a node that restarts to rejoin the mesh network automatically.	133
B.5	Raspberry Pi Docker installation. For the current non-root user to access Docker, it should be added to the <i>docker</i> group and the node should be restarted.	133
B.6	Docker Swarm configuration.	133
B.7	Environment variables required to run Apache Kafka and Zookeeper on a Raspberry Pi.	135
B.8	Application stack definition in docker-compose YAML: 1 of 1.	137
B.9	Application stack definition in docker-compose YAML: 2 of 11.	137
B.10	Application stack definition in docker-compose YAML: 3 of 11.	138
B.11	Application stack definition in docker-compose YAML: 4 of 11.	139
B.12	Application stack definition in docker-compose YAML: 5 of 11.	139
B.13	Application stack definition in docker-compose YAML: 6 of 11.	140
B.14	Application stack definition in docker-compose YAML: 7 of 11.	140
B.15	Application stack definition in docker-compose YAML: 8 of 11.	141
B.16	Application stack definition in docker-compose YAML: 9 of 11.	141
B.17	Application stack definition in docker-compose YAML: 10 of 11.	141
B.18	Application stack definition in docker-compose YAML: 11 of 11.	141

List of Tables

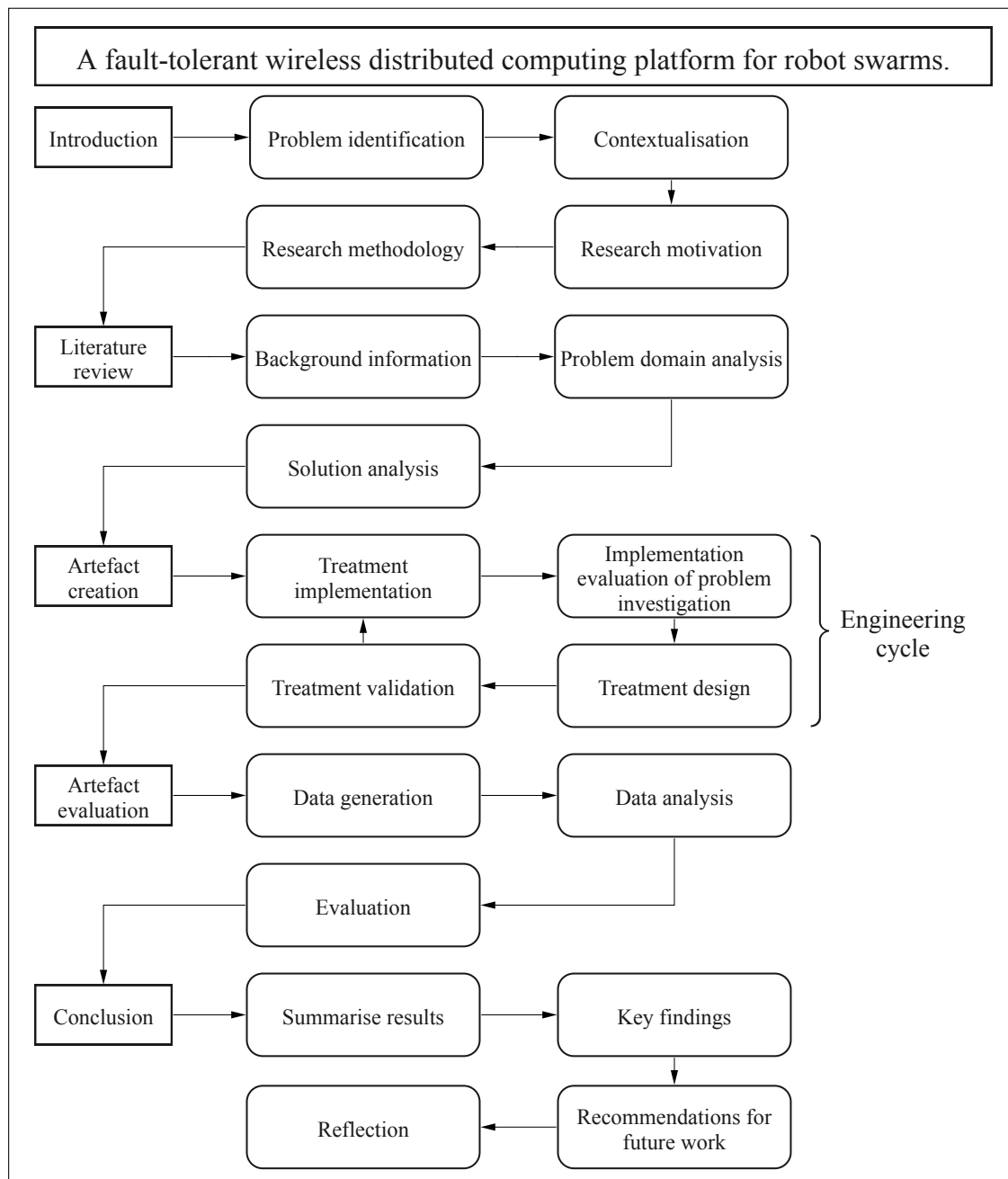
1.1	QoS requirements mapping [19].	6
1.2	Distinct classes of artefacts in Design Science Research [38].	12
1.3	Evaluation methods for DSR Research [38].	13
1.4	Mapping between type of DSR artefact and evaluation methodology used [38].	13
2.1	Different meanings of reliability in a distributed computing context [14].	16
2.2	Differentiation between system faults, errors and failures [39].	16
2.3	Classification of faults [40].	17
2.4	Classification of advanced fault-detection models [44].	19
2.5	Layer comparison between OSI and TCP/IP reference models [8].	21
2.6	Network modes [18].	23
2.7	Flynn's taxonomy [68].	30
2.8	Approaches to parallelism [75].	33
2.9	A direct comparison between virtual machines and containers [104].	47
3.1	Requirements set for selecting the hardware platform on which the artefact will be built.	54
3.2	State management in various situations using WebSockets.	61
3.3	Delivery semantics available in Apache Kafka [129].	70
3.4	Advantages of the streaming implementation over the RPC implementation.	71
3.5	Capabilities added by NATS streaming over NATS [134].	73
3.6	Description of sessions presented in Figure 3.18 on page 75. The sessions are grouped by which streams operator is running on them. Figure 3.19 on page 76 shows the combination of these operators to form the distributed fault-tolerant application.	76
3.7	Benefits of containerisation in distributed application development.	78
3.8	Summary of the layered platform architecture. References to the section that informed each layer's design choices are provided with descriptions of the layer and how it contributes to the fault tolerance of the platform as a whole.	81
4.1	Summary of mean (μ) and standard error (σ) results for the QoS metric measurements comparing infrastructure and mesh networking modes.	88

A.1 The number of processing cores in various computing devices. 124

A.2 In-depth comparison between Clusters, Grids and Clouds [71]. The categories most relevant to this research are reliability and loose coupling. 125

B.1 Notable command-line parameters presented by the NATS Streaming Server [133]. . 136

Research Map



The research map is presented to serve as a high-level overview of the dissertation. On the left, square blocks indicate the chapters while rounded blocks to their right indicate the themes relevant to the chapter. Arrows between the blocks indicate the sequence of the themes. The themes within each chapter are repeated at the start of the relevant chapter.

Chapter 1

Introduction

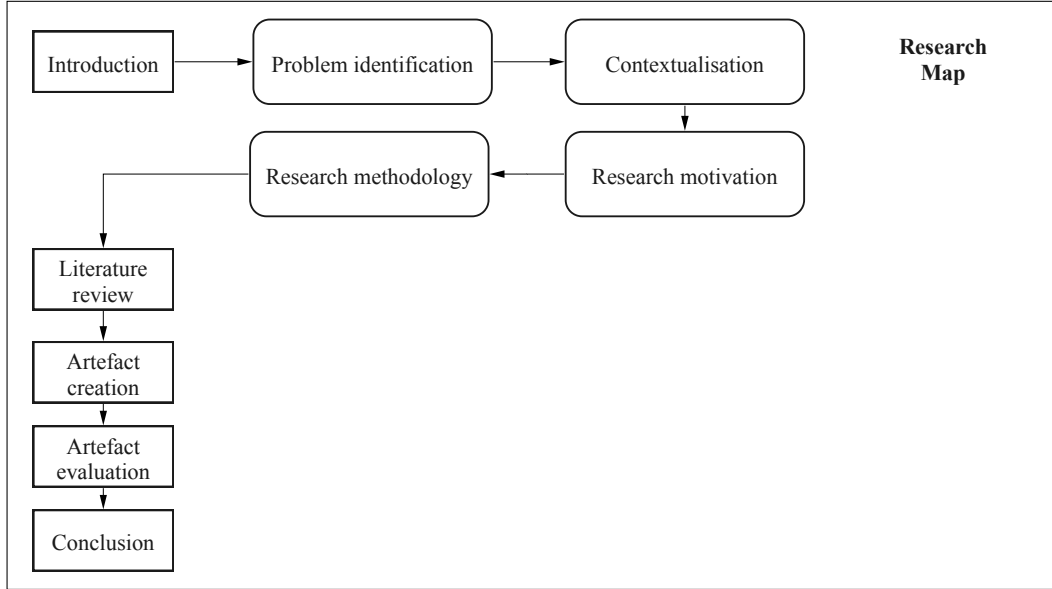


Figure 1.1 Research map extract for Chapter 1. The complete research map is presented on page 1.

At the 2018 Winter Olympic Games in PyeongChang, South Korea, Intel presented the first large-scale demonstration of swarm robotics [1]. The live demonstration included 300 quad-rotor Intel Shooting Star “drones” that flew in precise formations. A record-breaking 1218 drones were used in a separate recorded flight for broadcast purposes [2]. The changing formations created three-dimensional images in the sky, such as the snowboarder example presented on the left in Figure 1.2 on the following page. Drones are often referred to as unmanned aerial vehicles (UAV) or unmanned aerial systems (UAS). A UAV is defined as “*an aircraft piloted by remote control or on-board computers*” [3] and UAS is an extension that includes remote piloting or automation aspects.

The Intel drone demonstration was perhaps the public’s first large-scale introduction to robot teams and showed how many comparatively simple devices working towards a common goal could yield impressive results. The system that supported this display depended on a central control centre, shown on the right in Figure 1.2 on the following page. It broadcast a generated model of the desired flight paths, which contained the information every drone needed to maintain the correct formation over time. Each drone reported its current position, trajectory, percentage battery remaining and other information needed for generating the next model. While this approach proved effective for the Intel demonstration, it depends on a centralised control mechanism - a single point of failure. If a system fault occurred at this control centre and was not mitigated, the presentation would likely fail. An alternative approach in which the robot team’s coordination and computation are

decentralised could yield a more robust system [4].



Figure 1.2 *Left:* Intel drone swarm at the 2018 Winter Olympics opening ceremony [5]. *Right:* Intel drone team controlling a similar demonstration from a centralised control centre [6].

In *distributed systems*, computation is performed on multiple distinct devices, referred to as *nodes*, that contribute a portion of their processing power to the system [7]. In general, each node in a distributed system receives a logical sub-task of the system’s overall processing goal, completes the task, and returns the result to a control unit that assembles the intermediate results to produce the final result. The specifics of how each system performs distributed computing separate them into classes including grids, clusters and cloud-computing. All of these systems require networking infrastructure to enable nodes to coordinate their collective effort. In order to maximise the efficacy of the system, the network infrastructure must facilitate robust communication.

Networks, including computer, road and water distribution networks, are considered robust to the degree that individual component failures do not impact the functioning of the system as a whole. A system with a single point of failure cannot be considered robust because if it is exploited by an attacker or is subject to a random hardware failure, the entire system could grind to a halt. ARPANET, the first wide area network (WAN) and predecessor of the world-wide-web was constructed under the direction of the US department of defence to maintain a robust electronic communication channel that would survive an attack from the Soviet Union [8]. Eventually, ARPANET grew to include many university networks and government facilities using leased phone lines. Decentralising the network control and architecting for failure increased fault tolerance, which resulted in a network that could be expected to survive a nuclear attack.

Despite wireless networks being neither as reliable nor as fast as wired networks [9], there is a demand for wireless distributed computing using low-powered nodes [10], [11]. The combination of distributed computing, fault-tolerant systems and wireless networking presents an exciting challenge: could a fault-tolerant wireless distributed computing platform be constructed using simple and low-power components? If so, could such a platform provide communication and processing to support a drone swarm application? This is the challenge that this research explores.

In this chapter, the central concepts that form the research context are presented to establish the focus of the research. This focus guides the formulation of a central problem statement. The

research application area influences the scope of the research and guides the formulation of the aims and objectives: a high-level plan for solving the problem within a chosen application area. This chapter concludes with introductions to the research paradigm and the approach used to solve the research problem.

1.1 Contextualisation

In this section, the central theoretical concepts are introduced: robustness, wireless networking and distributed computing. These topics are presented in more detail in Sections 2.1, 2.2 and 2.3.

1.1.1 Robustness

The concept of robustness has several definitions, including “*being able to withstand or overcome adverse conditions*” [12] and “*being capable of performing without failure under a wide range of conditions*” [13]. A robust network will, therefore, if possible, not fail in meeting the demands of its users. One approach towards improving robustness could include eliminating single points of failure. This might produce a theoretically robust network while a more pragmatic view could include the characteristics that support the intended goal of the network [14]. In such a view, service demands are included in the evaluation of network robustness. The demands vary by the kind of service provided, but can be generalised to characteristics, such as:

- *Reliability* - how often active transmissions succeed in reaching their intended recipient;
- *Speed* - the amount of information that can be transferred in a given time; and
- *Availability* - how often the communication channel is accessible to senders and recipients of the information.

Real-world implementations of networks must manage compromises between reliability, speed and availability to support a robust platform. Furthermore, some implementations could have additional demands, such as network security, privacy or scalability [14]. Robustness, therefore, cannot be determined by a single litmus test. It requires the application of system architecture principles and analysis within the multiple requirements that the solution needs to fulfil [15].

1.1.2 Wireless networking

Wireless networking typically refers to a wireless local area network (WLAN) that uses radio waves for the transmission of data between devices. An ubiquitous implementation of wireless networking is a “*Wi-Fi*” network, a colloquial name referring to the IEEE 802.11 standard for networking in the 2.4 GHz and 5 GHz spectrum. Wireless networks enable connectivity between devices in situations where wired networks are impractical, such as on mobile devices [16], [17].

Computer networks support applications, such as email, internet access, resource sharing, audio and video streaming, video conferences, e-commerce and online gaming [18]. The network requirements vary greatly between these applications as a result of the differences in what constitutes a satisfactory

user experience. As an example, sending/receiving emails on a bandwidth-limited network can still be a satisfactory experience, whereas live video streaming on the same network could be a frustrating experience. These network requirements of applications can be expressed in terms of four factors that are used to describe network performance [8]:

- *Bandwidth* - the ability to move a volume of data during a unit of time;
- *Latency* - a measure of the time it takes a network packet to traverse the network;
- *Jitter* - the variation in packet latency between time intervals; and
- *Packet Loss* - the percentage of packets that do not arrive at their destination or arrive with errors.

These factors are referred to as the Quality of Service (QoS) metrics of a network. The earliest services provided over the internet only transmitted text and other simple data structures, and therefore connection reliability was the only concern. As soon as networks and the internet were used for more elaborate applications, such as multimedia, other factors became important. For analysis and comparison, the applications of networks can be classified in terms of:

- the type of data being transmitted;
- the quality of service demands that the application requires from the network; and
- the degree of tolerance to transmission errors.

The International Telecommunication Union (ITU) published recommendations for multimedia network applications in terms of their QoS requirements from an end-user perspective. Multimedia applications include the transmission of [19]:

- *Audio* - conversational voice, voice messaging and streaming audio;
- *Video* - videophone and one-way video; and
- *Data* - web browsing, file transfers, transaction services, command or control systems, interactive games, instant messaging, etc.

The ITU classifies applications as either error tolerant or error intolerant, referring to the applications' tolerance of packet loss. Example applications of multimedia transmission over networks are mapped in terms of their delay requirements and error tolerance, as shown in Table 1.1. Even when only considering a single QoS metric, it is evident that the network requirements vary significantly between different network applications.

	Interactive (Delay <1s)	Responsive (Delay ~2s)	Timely (Delay ~10s)	Non-critical (Delay >10s)
Error tolerant	Conversational voice and video	Voice/video messaging	Streaming audio and video	Fax
Error intolerant	Control systems	Transactions	Messaging, Downloads	Background, e.g. Usenet

Table 1.1 QoS requirements mapping [19].

An example of an error-intolerant application is a file transfer, where an unmitigated loss of connectivity (packet loss) renders the application unusable as the file will be corrupted. A cause of packet loss in networks is corruption by bit error, which could be caused by increased noise levels in the signal, momentary interference or even a malicious entity or attacker [20]. For the network to meet the demands of the file-transfer application, this problem needs to be overcome. The transmission control protocol (TCP) is a network transport protocol that is typically used in scenarios where data integrity is key, depending on the network scale [21]. TCP is intended to provide a “*reliable end-to-end byte stream over an unreliable network*” and is able to mitigate packet loss by [8]:

- establishing a transmission path from the sender to the receiver using a three-way-handshake before transmission begins;
- requiring the acknowledgement of sent packets’ reception before the next packets are sent; and
- re-transmitting unacknowledged packets after the retransmit timeout (RTO).

In another application, such as live video streaming of a sports event or concert to mobile devices, the demands on the network are quite different. In this application, the reception of the latest available data and adapting to changing network routes are prioritised higher than the integrity of the individual packets or the order in which they arrive. The robustness of this network application is determined by different factors than in the previous file-transfer example. For such an application, the user datagram protocol (UDP) is expected to perform better [22].

Network applications do not only vary in their tolerance for transmission errors, but also in their tolerance for network latency. Wireless networks not only present significantly higher latency than wired networks; they are also much more sensitive to the environment, which can make latency vary greatly. Real-time applications (RTA) require very low latency transmissions to be functional [23]. These applications have deadlines in which tasks must be completed. The acceptability or usefulness of a result received after the deadline classify them as hard, soft or firm real-time systems [24]. The intervals between deadlines are dictated by the environment in which the system must function. In so-called hard real-time systems, late results have zero utility, and the consequences of missing a deadline are catastrophic. An example of a hard real-time system is the monitoring system in a nuclear power plant. Such systems typically consist of specialised hardware and software and are costly [25]. In firm real-time systems, late results still have no utility, but do not result in a catastrophic failure. An example of such a system is a video conference. In soft real-time systems, late results have some utility, but multiple missed deadlines will degrade the application’s performance, for example, online transaction systems [24].

1.1.3 Distributed computing

Distributed computing is defined as “*a system where cooperating nodes communicate and coordinate their actions by passing messages*” [10]. A subset of this field includes high-performance computing clusters (HPCC) which are used to process large tasks quickly by distributing the processing

workload over many processors on multiple machines. In typical HPC implementations, processing nodes in the compute cluster coordinate their processing by passing messages through a high-speed network backbone. Since limitations in bandwidth or increased latency are particularly detrimental to cluster performance [26], one could say that the QoS requirements of this application are high.

To optimise efficiency, data centres often employ technologies, such as 100 Gigabit Ethernet or Infiniband, to support such systems. This high-performance network is intended to prevent a bottleneck in which processing resources are underutilised while waiting for task data to be transmitted. Distributed computing is, however, not confined to data centres and the enterprise. There is an increase in demand for distributed computing at edge locations (so-called fog computing) and IoT environments [27], [28] where wireless networking is often a requirement.

Wireless distributed computing (WDC) solutions are deployed in environments where a wired inter-connect is not feasible, for example, when nodes are not fixed in place. Wireless networks are unfortunately not as reliable as wired networks and are subject to random node failures, varying channel quality, stochastic availability of computing resources and unpredictable delays in message transit times [11]. Combining the fields of distributed computing and wireless networking creates challenges that the WDC field aims to solve.

1.2 Research and application areas

This research is divided into two logical components: the research area and the application area. The research area includes:

- Fault tolerance (presented in Section 2.1);
- Wireless networking (presented in Section 2.2);
- Distributed computing (presented in Section 2.3); and
- System architecture (presented in Section 2.4).

The chosen application area for this research is swarm robotics, presented in Section 2.5. It is not the goal to create a functional robot swarm, but rather to build a wireless distributed computing platform that is at least capable of supporting an application relevant to robot swarms. It is entirely possible and desirable that the artefact of this research could support applications beyond swarm robotics.

Scholarly literature at the intersection of these research concepts and the application area was found to be underexplored. Notable research contributions that have a focus close to that of this research include the works of Li and Shen that used wireless sensor networks (WSN) to control robot swarm behaviour [29]. Mesh networks were used as a network backbone in Nembrini's work on minimalist coherent swarming of mobile robots [30]. Neither of these works nor other tangential research encountered had a specific focus on fault tolerance and wireless distributed computing in robot swarms. This research aims to shed light on this under-explored subject and support future research contributions.

1.3 Problem statement

The problem that this research project aims to address is that of creating a fault-tolerant platform that is suitable for distributed computing, using wireless networking for communication. The solution implements established system architecture patterns to enhance its robustness through fault-tolerance. Figure 1.3 illustrates how the three central theoretical concepts support the solution.

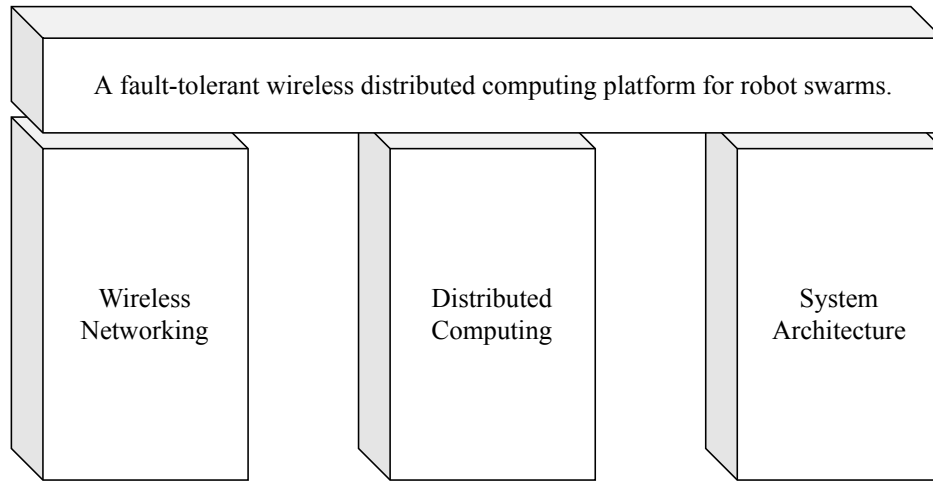


Figure 1.3 The theoretical fields supporting this research.

Problem statement: How can existing system architectures be used to create a fault-tolerant wireless distributed computing platform?

1.4 Aim and objectives

The aim of this research is to apply system architecture principles to create a fault-tolerant platform capable of supporting wireless distributed computing for a robot swarm application. The focus of the research is not to address the complexities of robot swarms operating in physical spaces. Instead, the swarm can be simulated in software and given an appropriate task to solve using distributed computing. To achieve this aim, the following objectives are set:

1. Investigate literature on the central research concepts of fault tolerance, wireless networking, distributed computing and system architecture;
2. Design a simple robot swarm application to guide the development of a platform to support it;
3. Establish a baseline instantiation of a wireless distributed computing platform;
4. Perform iterative prototyping on the platform to increase its fault tolerance, evaluating each iteration, using the application as an illustrative example;
5. Evaluate the final prototype's performance; and
6. Summarise results and present the conclusions.

1.5 Research paradigm

A research paradigm is defined by an “*interrelated set of assumptions about the social world which provide a philosophical and conceptual framework for the organised study of that world*” [31]. The paradigm chosen for this research is the *positivistic* paradigm.

Positivism is characterised by the view that the world is ordered, not random, and that we can observe it objectively. Repeatability is an important concept within this paradigm. For example, in a defined and controlled environment, such as an experiment, applying the same actions should yield the same results. It emphasises the refutation or confirmation of previously held knowledge as a process to improve the quality of knowledge [32].

The nature of reality (ontology) of positivism is based on David Hume’s view that reality consists of atomistic and independent events [33]. Due to the technical nature of this research, it naturally fits into a quantitative research method with empirical research: “*all phenomena can be reduced to empirical indicators which represent the truth*” [34].

The theory of knowledge (epistemology) associated with positivism is based on the belief of René Descartes that reasoning with a deductive method is the best way to generate knowledge about reality [33]. Designing and creating a research artefact, experimenting with it and applying a deductive method on the experimentation results, will generate insight to be communicated.

1.6 Research approach

The chosen research approach is design science research (DSR). DSR is oriented to the creation of successful artefacts [35]. Weber [36] found that DSR can be used in combination with multiple paradigms, including positivism, interpretivism and developmentalism. It has also been used successfully in combination with other approaches, such as ethnography and behavioural science [31]. DSR is conducted through iterations of solving design problems and answering knowledge questions through the creation of artefacts. DSR has its roots in the engineering discipline and therefore includes the use of the engineering cycle, as shown in Figure 1.4 on the following page.

1.6.1 Steps and guidelines

The tasks in the engineering cycle correspond to the generalised steps of the DSR approach [35] and guides the structure of this research:

1. Problem identification and motivation (Chapter 1);
2. Definition of objectives for a solution (Chapter 2);
3. Design and development (Chapter 3);
4. Demonstration (Chapter 3);
5. Evaluation (Chapter 4); and
6. Communication (Chapter 5)

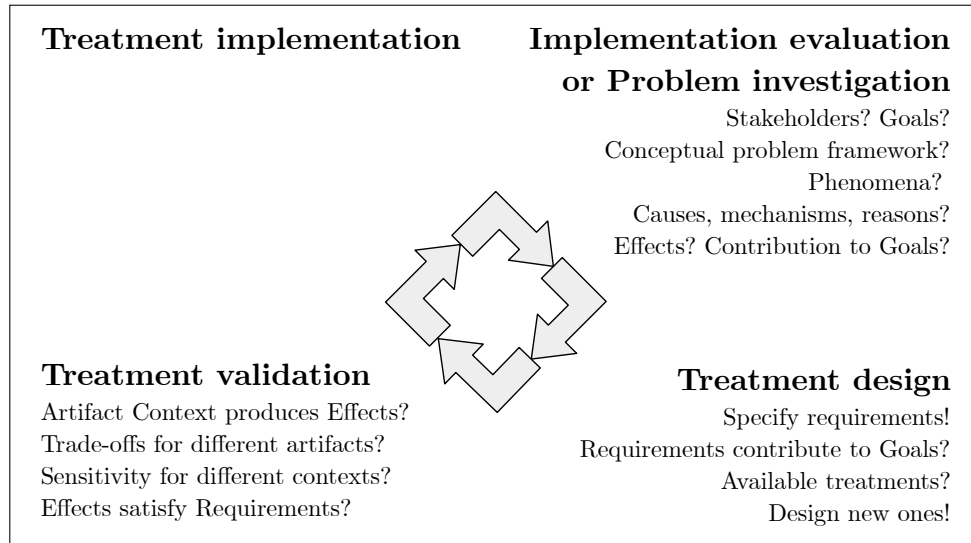


Figure 1.4 The engineering cycle [36]. Question marks indicate knowledge questions and exclamation marks indicate design problems. The first iteration of the cycle starts at *Problem investigation*.

The engineering cycle as presented in Figure 1.4 is especially relevant in Step 3, where multiple short iterations of it are performed. This focus is indicated in the Research Map on page 1.

While the steps are to be followed in sequence, the DSR approach prescribes seven research guidelines, which should be adhered to throughout [37]:

- *Design as an artefact* - a viable construct, model, method or instantiation;
- *Problem relevance* - developing technology-based solutions to business problems;
- *Design evaluation* - demonstrable utility, quality and efficacy of the artefact;
- *Research contributions* - must provide research contributions in the areas of the design artefact;
- *Research rigour* - application of rigorous construction and evaluation methods of artefact design;
- *Design as a search process* - search for an effective artefact to reach desired ends while satisfying laws in the problem environment; and
- *Communication of research* - research must be presented effectively to both technology- and management-oriented audiences.

The result of DSR is, by definition, a purposeful artefact created to address a problem [37]. Therefore, the most important characteristic by which research conducted with this approach is judged, is that it must produce an “*artefact created to address a problem*”. An artefact is “*any designed object with an embedded solution to an understood problem*” [35]. The artefact of this research is a fault-tolerant platform consisting of hardware, software and configuration working harmoniously to support wireless distributed computing. Producing such an artefact requires the identification and investigation of problems in these domains so that they can be solved by applying system architecture principles in the creation of the artefact.

1.6.2 Research outputs

In keeping with the first guideline of DSR, “*design as an artefact*”, this research produces artefacts that embody the knowledge gathered through the research process. The sixth guideline, “*design as a search process*”, defines the artefact creation process presented in Chapter 3. Peffers et al. [38] identified distinct types of research artefact in DSR, presented in Table 1.2.

Artefact Type	Description
Algorithm	An approach, method, or process described largely by a set of formal logical instructions
Construct	Concept, assertion, or syntax that has been constructed from a set of statements, assertions, or other concepts
Model	Simplified representation of reality documented using a formal notation or language
Framework	Meta-model
Instantiation	The structure and organisation of a system’s hardware or system software or part thereof
Method	Actionable instructions that are conceptual (not algorithmic)

Table 1.2 Distinct classes of artefacts in Design Science Research [38].

Of these outputs, this research includes:

1. an *instantiation* of a fault-tolerant wireless distributed computing platform; and
2. the *method* developed to arrange components of the artefact in such a way as to support fault-tolerant wireless distributed computing.

The instantiation of the platform will be evaluated with an appropriate method, described in the next section.

1.6.3 Evaluation methods

Peffers et al. [38] reviewed a selection of 148 journal articles that implemented DSR. The methods used to evaluate the outputs of the research were tabulated and are presented with their descriptions in Table 1.3 on the following page. In addition to identifying distinct evaluation methods, Peffers et al. mapped the connections between different types of artefact and the evaluation methods used. This mapping is presented in Table 1.4 on the following page. The technical experiment is also the most popular evaluation method for both of the chosen artefact types presented in Section 1.6.2. The evaluation of the artefact is therefore performed using two methods:

1. The *illustrative scenario* method is used iteratively to guide the development of the artefact in Chapter 3; and
2. The *technical experiment* method is used in Chapter 4 to determine the performance of the artefact.

Evaluation method	Description
Logical argument	An argument with face validity
Expert evaluation	Assessment of an artefact by one or more experts
Technical experiment	A performance evaluation of an algorithm implementation, using real-world data, synthetic data, or no data, it is designed to evaluate the technical performance, rather than its performance in relation to the real world
Subject-based experiment	A test involving subjects to evaluate whether an assertion is true
Action research	Use of an artefact in a real-world situation as part of a research intervention, evaluating its effect on the real-world situation
Prototype	Implementation of an artefact aimed at demonstrating the utility or suitability of the artefact
Case study	Application of an artefact to a real-world situation and evaluating its effect
Illustrative scenario	Application of an artefact to a synthetic or real-world situation aimed at illustrating suitability or utility of the artefact

Table 1.3 Evaluation methods for DSR Research [38].

	Logical argument	Expert evaluation	Technical experiment	Subject-based experiment	Prototype	Action research	Case study	Illustrative scenario	None	Total
Algorithm	1	-	60	1	-	-	-	3	-	65
Construct	3	-	3	2	2	-	-	2	-	12
Framework	1	1	-	-	1	-	1	4	1	9
Instantiation	-	-	5	1	1	-	-	1	-	8
Method	2	-	14	4	-	-	7	6	-	33
Model	3	-	10	-	2	2	-	4	-	21
Total	10	1	92	8	6	2	8	20	1	<i>148</i>

Table 1.4 Mapping between type of DSR artefact and evaluation methodology used [38].

1.7 Summary

In this chapter, concepts relevant to the research were introduced in Section 1.1 to contextualise the research. A distinction was made between the research area and application area in Section 1.2. The

research problem was identified in Section 1.3 and aims and objectives were formulated to address it in Section 1.4. The chapter covered the chosen research paradigm and approach to apply to the research problem in Sections 1.5 and 1.6. The research approach includes steps and guidelines, the planned outputs of the research and the evaluation methods used. In the next chapter, the existing literature is reviewed to provide more in-depth information on the concepts relevant to the identified problem.

Chapter 2

Literature Review

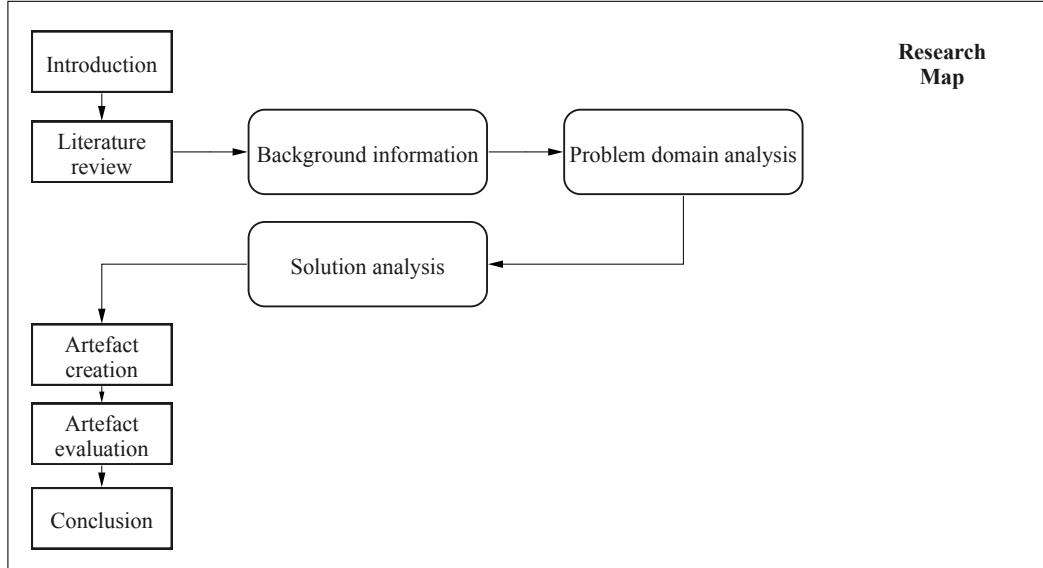


Figure 2.1 Research map extract for Chapter 2. The complete research map is presented on page 1.

The aim in this chapter is to review and present existing literature that relates to the research. Background information is provided, the research contribution is placed in context with other works and the substantiation of the artefact of this research is aided. In order to keep the narrative of chapter focussed, figures and tables that are non-essential to view while reading the chapter are presented in Appendix A on page 123. Referring to the research map, the first theme in this chapter is background information. Background information for the central theoretical concepts is presented in Sections 2.2 to 2.4. In this chapter, the focus is on both the problem domain analysis and solution analysis that are required to develop a successful platform.

2.1 Platform robustness

The concept of robustness is introduced in Section 1.1.1. In this section, this introduction is investigated in greater depth. The term robustness is often associated with reliability, but as Birman [14] remarks, it can have many meanings, depending on the context. In the distributed computing system context alone, it has several meanings, summarised in Table 2.1 on the following page.

While noting the importance of all the different meanings of reliability in the context of distributed computing systems, this research is focused on enhancing platform robustness through increased fault tolerance [14], as indicated in the research title. The platform of this research refers to a collection of hardware and software in a specific configuration that supports wireless distributed

Meaning	Description
Fault tolerance	The ability to recover from component failures without performing incorrect actions
High availability	The ability to continue providing services during periods when components have failed
Continuous availability	The ability to provide uninterrupted service to its users
Recoverability	The ability of failed components to rejoin the system after a failure has been repaired
Consistency	The ability to coordinate related actions performed by multiple components
Scalability	The ability to continue correct operation after some aspect is scaled to a larger size
Security	The ability to protect data, services and resources against misuse by unauthorised users
Privacy	The ability to protect sensitive data from unauthorised disclosure
Correct specification	The assurance that the system solves the intended problem
Correct implementation	The assurance that the system correctly implements its specification
Predictable performance	The ability to guarantee that the distributed system achieves the desired levels of performance
Timeliness	The assurance that actions are taken within specified time bounds

Table 2.1 Different meanings of reliability in a distributed computing context [14].

computing. The design and implementation of the platform must be sturdy to be able to perform without failure in adverse conditions. Kumari and Kaur [39] distinguish between system faults, errors and failures, as presented in Table 2.2.

By the time a user is presented with a system failure, the system has moved to an error state caused by a system fault. The prevention and/or mitigation of system faults is therefore critical to producing reliable systems [14]. It is also possible for system faults to occur without presenting an observable failure to the user. In this section, system faults, fault detection mechanisms and fault-tolerance approaches are presented.

Term	Description
Faults	The inability of a system to perform a necessary task, caused by an abnormal state or software bug in one or more system components.
Errors	A system component moves to an error state due to the presence of a fault.
Failures	The misbehaviour of a system as observed by a user. Failures are only recognised when the system's output/outcome is visibly incorrect.

Table 2.2 Differentiation between system faults, errors and failures [39].

2.1.1 System faults

System faults can be classified into two broad categories [40]: crash faults and byzantine faults. These categories are described in Table 2.3 and examples of system faults that belong to the categories is presented in Figure 2.2.

Fault Category	Description
Crash fault	These faults are caused by a failure of one or more system components, e.g. processors, disks, power supplies, etc. The faults occur at a hardware level and as such often require manual interaction, but software systems can be developed to tolerate these faults.
Byzantine fault	These faults occur when there is ambiguity in the system, e.g. a component can appear both failed and functional to different observers. Such a fault affects the system at a logical level and does not require manual interaction.

Table 2.3 Classification of faults [40].

Complex systems are often structured as layers, each of which depends on the layer below and exposes abstracted functionality to the layer above. Faults can occur at different levels, such as the conceptual physical, platform and service layers. A fault at the physical layer is therefore likely to affect the layers above as the services it provides could be unavailable [40].

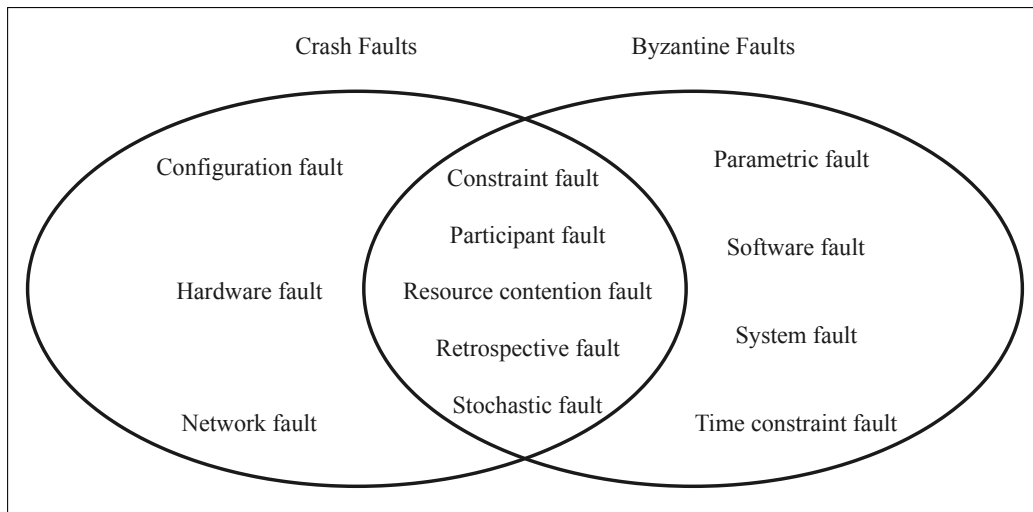


Figure 2.2 Categorisation of faults in cloud computing environments [40].

As an example, consider a power supply unit that fails unexpectedly. This fault is at the physical layer, and there is nothing that the operating system at the platform layer is able to do to prevent the system from halting - it is a catastrophic failure. The catastrophic fault at this layer needs to be proactively mitigated at the same layer by installing a redundant power supply, such as the one presented in Figure 2.3 on the following page.

In other cases, it may be possible to mitigate against a physical layer fault at a higher layer of



Figure 2.3 Redundant power supply unit (PSU) for servers/workstations [41] that represents a proactive mitigation against a physical layer fault.

abstraction. As an example, consider a network fault that unexpectedly disconnects a server. This fault is also at the physical layer, but in this case, the operating system and running applications could cache the data that had to be transmitted in memory until the network fault has been resolved, preventing a catastrophic failure.

2.1.2 Fault-detection mechanisms

According to Tanenbaum and Van Steen [42], fault detection is one of the cornerstones of fault-tolerant distributed systems. They present two basic mechanisms for detecting process failures:

1. *Health check* - Actively send “are you alive?” messages to a process and test for a response. If the process stops responding to these messages, it is assumed that the process failed; and
2. *Heartbeat* - Continuously monitor messages from the process. If the messages stop, the process is assumed to have failed.

The inability to respond to ping messages or to broadcast availability due to unreliable networks poses a problem, as this does not always mean that a service is in a failed state. These approaches generate false positives and could lead to the incorrect activation of a fault-tolerance mechanism. In some distributed applications, such as wireless sensor networks (WSN), more sophisticated fault detection mechanisms should be implemented, e.g. self-diagnosis or cooperative diagnosis [43].

As a large portion of distributed systems built today depends on managed services provided by public cloud providers, fault detection becomes a greater challenge. In such complex distributed environments, Zhang et al. [44] present an approach using a support vector machine (SVM) to enable fault prediction. This approach leverages data models of two separate classes, presented in Table 2.4 on the following page.

Model class	Description	Examples
Rule-based	Models that discern a fault based on its characteristics	Signature methods, similarity judgments and decision trees
Statistics-based	Models that discern a fault based on the data collected from systems	Neural networks, learning vector quantisation, support vector machines

Table 2.4 Classification of advanced fault-detection models [44].

2.1.3 Fault-tolerance approaches

Being fault tolerant is defined as “*relating to or being a computer or program with a self-contained backup system that allows continued operation when major components fail*” [45]. Hasan and Goraya [40] reviewed fault tolerance in distributed cloud computing environments and proposed that some faults can be categorised as both crash faults and byzantine faults, as shown in Figure 2.2 on page 17. Approaches to improving fault tolerance can be categorised as either reactive or proactive. Reactive approaches are applied once a fault has occurred and has been detected while proactive approaches are applied before the fault occurs. Figure 2.4 presents a generalised view of these approaches with examples.

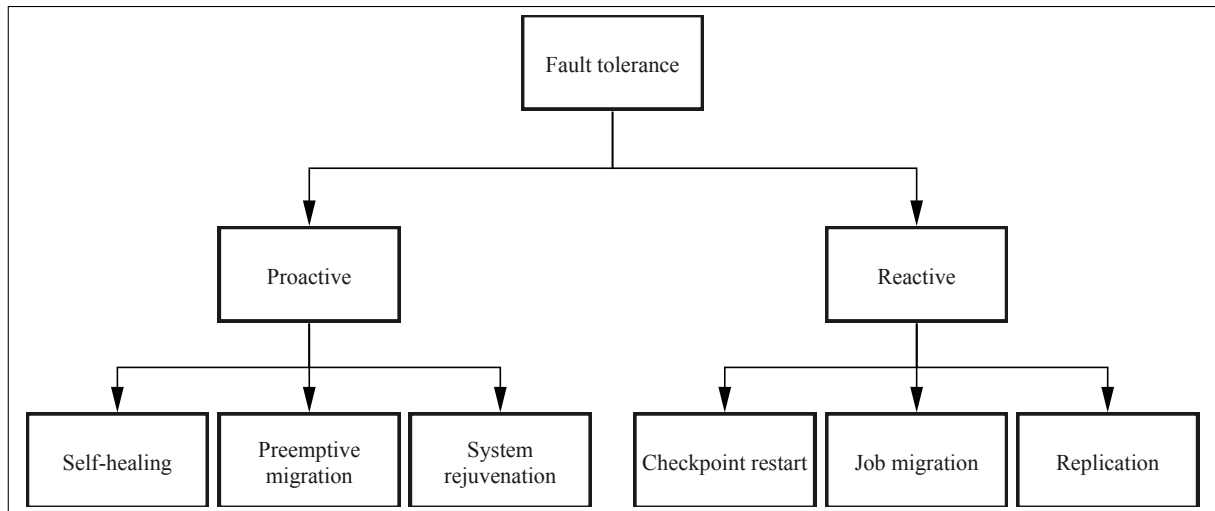


Figure 2.4 Fault-tolerance approaches in cloud computing environments [40].

The work of Hasan and Goraya [40] focused on cloud computing environments, whereas this research focuses on wireless networks, which are inherently localised. Nevertheless, the approaches presented in Figure 2.4 can be applied more generally, and are investigated individually in Sections 2.1.3.1 and 2.1.3.2 on the following page.

2.1.3.1 Proactive fault-tolerance approaches

In this section, the three proactive fault-tolerance approaches presented in Figure 2.4 are discussed. These approaches are applied in anticipation of a fault occurring to increase the fault tolerance of

the system [39].

Self-healing is defined as *“the capability of a system to have an autonomous recovery from faults by periodically applying specific fault-recovery procedures consisting of supervision tasks”* [40], [46]. A self-healing system can detect a change to a faulty state and restore to a normal state without human interaction.

Pre-emptive migration aims to predict faults, such as node failures, ahead of time and to migrate any running processes to another node before the node fails [40], [47]. An early indicator of node failure could include a sustained increase in heat from system components, such as a CPU or GPU [48]. This probabilistic approach is focused on using the failure rates of individual computing nodes rather than on the aspects of the fault itself.

System rejuvenation is the process of keeping the application/system state fresh [40]. It involves terminating the application, cleaning up its internal state and restarting it to prevent the occurrence of future failures [49]. So-called fixed-time rejuvenation performs the process at a set time interval, e.g. once a week at a time when system usage is expected to be low. Variable time rejuvenation is not performed on a predictable cycle. Instead, it depends on the working conditions of the system. Finally, the rejuvenation itself can be either partial, where a portion of the overall system is rejuvenated or full, where the entire system is rejuvenated. In distributed and heterogeneous environments, failure rates of individual components vary, hence partial rejuvenation at variable times *“looks to be of more use rather than full rejuvenation”* [40].

2.1.3.2 Reactive fault-tolerance approaches

In this section, the reactive approaches to fault tolerance are presented. These approaches do not need to examine system behaviour over an extended period of time, but can react quickly to a fault as soon as it is detected.

Checkpoint restarts involve periodically storing the state of the system or application [40]. In the event of a fault being detected, the system or application is restored from the last checkpoint that was stored [50]. The checkpoint restart approach can be applied both on its own or in combination with additional fault-tolerance approaches, making it a very popular approach. The approach lends itself to being optimised for a specific scenario, as the failure rates of system components can be used to determine how often checkpoints are stored.

Job migration is often used to mitigate against crash faults and unlike its proactive counterpart, pre-emptive migration, is only applied after the fault has occurred [40]. A failed task is migrated to a suitable resource. In this approach, the processing job needs to be transferable, and communication overhead needs to be considered [48].

Replication is the most popular [40] and involves running a task on multiple execution instances. In the case of a fault being detected, the replicated instances can continue to operate [39]. Replication is an effective approach against both crash faults and byzantine faults and can be active or passive.

Active replication involves the replication of processing jobs to both the primary and backup execution instances. In contrast, passive replication only replicates the jobs of the primary execution instance to backup instances. Only in the event that the primary instance fails, can a backup instance take over processing.

2.2 Wireless Networking

Wireless networks use radio waves rather than guiding materials, such as copper or optical fibre, to facilitate communication between devices [8]. A transmitter can send bits of data to a receiver's antenna that is within range. This grants wireless networking installations the flexibility to operate in dynamic environments where wired connections are not feasible. The increase in flexibility comes with limitations, including significantly higher signal degradation and interference from other transmissions [8]. In this section, wireless networking is investigated. The OSI reference network model provides a theoretical structure for analysing computer networks in general and is therefore presented first. Wireless networks based on the IEEE 802.11 specification are introduced in Section 2.2.2 with a focus on characteristics relevant to fault tolerance. Concepts relevant to evaluating network performance are presented in Section 2.2.5.

2.2.1 OSI reference network model

The Open Systems Interconnection (OSI) reference model was based on a proposal developed by the ISO (International Standards Organisation) to standardise protocols used in systems that are open for communication by other systems. It is a seven-layer model in which each layer uses services and functions from the layer directly below it, and in turn provides services and functions to the layer directly above it [8]. The OSI Reference Model is not an implementation of a network architecture, and the protocol implementations initially associated with it are no longer in use [8]. The TCP/IP model is, however, in wide use and a mapping of the layers is presented in Table 2.5. The OSI reference model is, however, still valid as a guide to understanding networks.

OSI		TCP/IP	
7	Application	Application	4
6	Presentation		
5	Session	Transport	3
4	Transport		
3	Network	Internet	2
2	Data Link	Link	1
1	Physical		

Table 2.5 Layer comparison between OSI and TCP/IP reference models [8].

Each of the seven layers within the OSI reference model are briefly introduced [8]:

1. The physical layer is responsible for transmitting raw bits over a transmission medium, such as an ethernet cable, fibre optic cable or wireless antenna. It is responsible for modulation

of data bits by defining the electrical, timing and other interfaces by which bits are sent as signals over channels;

2. The data-link layer provides an abstraction of the physical medium into a link that appears free from errors to the network layer. It is responsible for creating frames of data, checking received frames and correction of transmission errors received from the physical layer;
3. The network layer is responsible for controlling the operation of the subnet and determining how data packets are routed to their intended receiver. This routing enables transmission of data across separate networks by identifying devices by their logical address as opposed to the physical address (MAC). Routing tables are used to describe the extended topology of an extended network of networks;
4. The transport layer provides an abstracted connection stream from a process running on one machine to a process running on another machine. It provides two types of transport service: connectionless and connection-oriented, which correlate to network services provided by the network layer. The four bottom layers, physical to transport, can be referred to as the transport service provider, while the three remaining layers can be referred to as the transport service user. The transport layer is, therefore, in a significant position, as it is on the edge of the transport service provider, meaning it is the only layer to which applications are exposed. The two most common transport protocols used by internet services are TCP and UDP;
5. The session layer is responsible for establishing, managing and terminating connection sessions created between the presentation layer instances on two communicating computers. It provides services, including dialogue management, synchronisation, activity management and exception management. It is important to note that the TCP/IP protocol stack does not include a session layer [51];
6. The presentation layer is responsible for the syntax and semantics of the data transmitted between devices because different systems may use different internal data structures [8]. For example, different operating systems and applications use different character encodings. To facilitate communication between such systems, the presentation layer converts the data to an encoding format that will be understood on the receiving machine before it is sent; and
7. The application layer provides services to the user of the system and contains the protocols that the programs they use need to do tasks, such as sending an email, browsing the Internet, resource sharing, etc. [8].

The seven layers provide a useful framework within which network functionality and behaviour can be investigated. The research is, however, focused on the application of wireless networks to a specific problem. The ubiquitous IEEE 802.11 wireless networking standard is presented next.

2.2.2 IEEE 802.11 wireless networks

Wireless networks have a very broad range of implementations. Besides the mobile networks used by cellular phones, IEEE 802.11 wireless local area networks are the most widely deployed wireless networking applications [52]. The focus of this research will also be the application of IEEE 802.11

Network mode	Description
Infrastructure	The most widely used mode and is the mode used when a client device (e.g. smartphone, tablet or laptop) connects to a centralised access point. All transfers in this mode are managed by the access point. Clients communicate over a single hop with the access point which directs packets either to other clients in the local network or acts as a router to connect the client to an extended network, e.g. the Internet.
Ad Hoc	This mode is typically characterised by single-hop transmissions between clients without a base station or access point. However, clients could facilitate multi-hop communication by coordinating the transmissions of other nodes.
Mesh	In this mode, a network can either be infrastructure-based or infrastructure-less. In both cases, nodes that are within range of each other form multiple redundant routing paths. If the network includes a base station, the mesh could be connected to other extended networks. Examples of such networks are found in wireless sensor networks. If the network does not include a base station, nodes will simply relay messages towards each other based on a mesh routing protocol, but will not facilitate communication to external networks.

Table 2.6 Network modes [18].

“*Wi-Fi*” networks to support distributed computing.

The 802.11 and 802.11x family of specifications are developed by the IEEE for wireless local area networks (WLAN). Kurose and Ross [18] present a high-level classification between wireless network implementations. The two criteria in this classification are:

1. whether a packet in the wireless network crosses exactly one wireless hop or multiple wireless hops; and
2. whether or not there is any infrastructure, such as a base station, in the network.

Based on these classifications, three relevant network modes are summarised in Table 2.6.

Mesh networks are of particular interest to this research; they eliminate the single point of failure of an infrastructure endpoint. The redundant routing paths are expected to enhance the fault tolerance of a distributed system that uses it. Wireless networks based on the 802.11 specifications have become ubiquitous, but there are challenges associated with their use. Sections 2.2.3 and 2.2.4 are dedicated to two of these challenges.

2.2.3 Challenge 1: Crowded 2.4 GHz spectrum

Most consumer implementations of wireless networking use Wi-Fi (IEEE 802.11), Bluetooth (IEEE 802.15) and Zigbee (IEEE 802.15.4) networks [53]. This is primarily due to the unlicensed network bands in which they operate, known as Industrial, Scientific and Medical (ISM) bands [54]. These bands were made available by the International Telecommunications Union (ITU) [55] and vary by

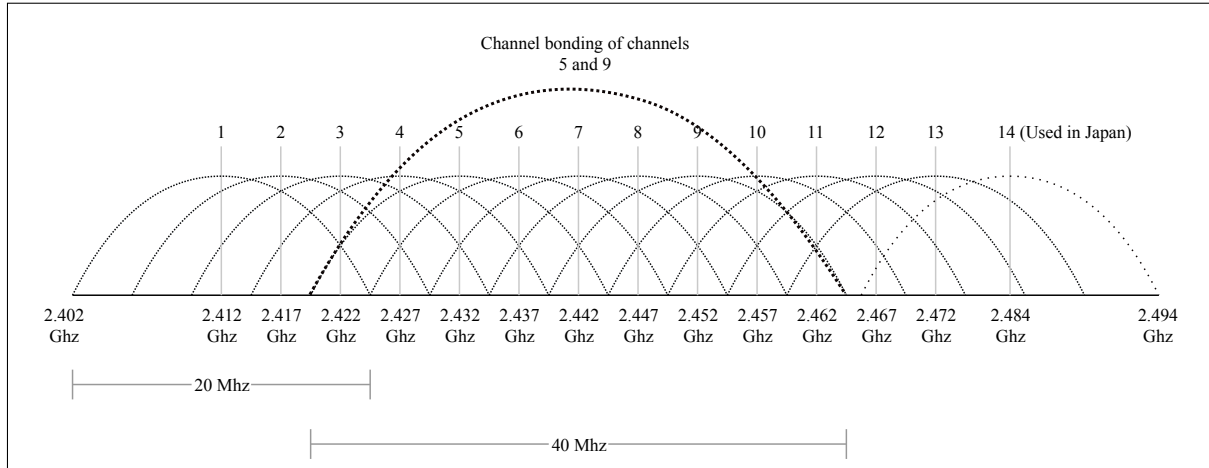


Figure 2.5 Representation of Wi-Fi channels with 802.11g/n 20 MHz channels and 40 MHz (802.11n only) “wide channels”. Figure adapted from representations by [57], [58].

region. Globally, the most commonly used frequency range within the ISM bands is between 2.4 and 2.5 GHz.

Wireless channels provide separation of device transmissions by splitting the available spectrum. In the case of IEEE 802.11n, the 100 MHz spectrum band between 2.4 and 2.5 GHz is split into 14 channels that are 20 MHz wide. Figure 2.5 shows the 14 channels with a bandwidth of 20 MHz each. Note that channel #14 is specifically used in the Japan region, whereas channels 1-13 are used globally. Since 13 channels of 20 MHz do not divide into the available 100 MHz spectrum, only channels 1, 6 and 11 are orthogonal, i.e. they do not overlap with each other [56].

IEEE 802.11n introduces support for channel bonding, which allows devices to use multiple channels simultaneously for increased throughput. In order to optimally use the available spectrum, devices periodically scan for the best available channel and switches to it. This approach is referred to as channel hopping.

Figure A.1 on page 123 shows an example of a mobile application¹ that can be used to visualise this information. The screenshots reveal how crowded the spectrum can be in environments where there are many separate networks, such as buildings with multiple apartments. Two of the detected access points shown in the figure are not typical access points that connect other devices. These are examples of Internet-of-Things (IoT) devices (a speaker system and a printer in this case) that take up a portion of the available spectrum for their limited function. Beyond the 802.11b/g/n Wi-Fi devices, other devices, including wireless computer peripherals, cordless phones and baby monitors also use the 2.4 GHz band. On-person devices often use the 802.15 personal area network (PAN) implementation known as Bluetooth, including wireless headphones, smart watches, fitness trackers and more. Many smart home products either directly access the 2.4 GHz Wi-Fi network (e.g. Sonoff B1 smart-bulb) or use a bridge that joins the home network to devices utilising the

¹<https://github.com/VREMSSoftwareDevelopment/Wi-FiAnalyzer>

low-power 802.15.4 Zigbee [53], [54] implementation (e.g. Samsung SmartThings hub and Philips Hue hub).

All of these devices have to share the same ISM band, and it is clear that there is not a wide enough spectrum for all the users to have a satisfactory wireless networking experience. Due to the severely crowded 2.4 GHz band, the more modern IEEE 802.11ac and upcoming 802.11ad standards utilise the 5 GHz and 60 GHz bands respectively. The breakdown of available bands within the 5 GHz spectrum range is presented in Figure 2.6. Given a fixed bandwidth, the number of channels within the band depends on the width of the channels. Where 802.11n supported only 20 MHz and 40 MHz through channel bonding, 802.11ac extends this to 80 MHz and 160 MHz.

			IEEE standard (bandwidth)			
Device support	Band	Center Frequency (MHz)	802.11a/b/g (20MHz)	802.11n (40MHz)	802.11ac (80MHz)	802.11ac (160MHz)
ALL	UNII-1	5180	36	38 [36-40]	42 [36-48]	50 [36-64]
		5200	40			
		5220	44	46 [44-48]		
		5240	48			
DFS	UNII-2	5260	52	54 [52-56]	58 [52-64]	
		5280	56			
		5300	60	62 [60-64]		
		5320	64			
	UNII-2e	5500	100	102 [100-104]	106 [100-112]	114 [100-128]
		5520	104			
		5540	108	110 [108-112]		
		5560	112			
		5580	116	118 [116-120]		
		5600	120			
		5620	124	126 [124-128]		
		5640	128			
		5660	132	134 [132-136]	138 [132-144]	
		5680	136			
		5700	140	142 [140-144]		
		5720	144			
ALL	UNII-3	5745	149	151 [149-153]	155 [149-161]	
		5765	153			
		5785	157	159 [157-161]		
		5805	161			
	ISM	5825	165			

Figure 2.6 Channels available in the Unlicensed National Information Infrastructure (U-NII) bands within the 5 GHz spectrum. Channel bonding is applied in IEEE 802.11n and 802.11ac to simultaneously use multiple standard 20 MHz wide channels to support increased throughput. Figure adapted from work by [58].

At the widest frequency band of 160 MHz, there are only two channels available within the 5 GHz spectrum. Furthermore, only devices supporting Dynamic Frequency Selection (DFS) are able to use the U-NII-2 and U-NII-2e bands. Despite these challenges, the increase in the number of channels at various channel widths combined with the ability of modern devices to quickly switch between frequencies using DFS presents a significant step forward in mitigating the challenges of the crowded 2.4 GHz spectrum.

2.2.4 Challenge 2: High packet loss rate and ossified transport protocols

Many of the challenges associated with wireless networking are a result of two characteristics: transmission errors occur very frequently, and channel capacity is variable [8]. Due to external interference and other factors, a typical packet loss rate can exceed 10%. For comparison, a loss rate for wired connections of 1% is considered moderate. The applications that use networks still need to function in these conditions, so it is the responsibility of network transport protocols to deal with lossy connections. The most common transport protocol intended to provide reliable connections is the transmission control protocol (TCP). One of the TCP strategies that promote reliability is to apply congestion control algorithms to prevent a fast sender from flooding a slow receiver [8]. As an example of some of the challenges that high packet loss rate and variable channel capacity can cause, consider the challenge created in the following example:

- ARQ (Automatic Repeat reQuest [8]) protocols at the Data Link Layer will retransmit data frames that are unacknowledged (assumed lost); and
- A reliable transport protocol (e.g. TCP) will apply congestion control when it thinks the sender is faster than the receiver.

Both these functions are good and necessary for reliable networks, but when many packets are dropped, it can cause the TCP protocol to apply congestion control incorrectly. The TCP protocol cannot know that its route contains unreliable links, because it operates at the network layer and is simply presented with a stream of frames by the data link layer (refer to Section 2.2.1). As such, it cannot tell the difference between packet loss caused by a lossy wireless network and a slow receiver that is not able to acknowledge the reception of packets. When this happens, it dramatically reduces the efficiency of transmission, since even on fast wireless networks, the transmission rate gets throttled [8]. Padhye et al. [59] present a formal approximation of the relationship between the packet loss rate and throughput when TCP is used. This relationship is presented in the approximation 2.1:

$$B(p) \approx \min\left(\frac{W_{max}}{RTT}, \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_0\min(1, 3\sqrt{\frac{3bp}{8}})p(1 + 32p^2)}\right) \quad (2.1)$$

Where:

- B is the connection throughput;
- W_{max} is the maximum congestion control window size;
- RTT is the average round-trip time as measured by TCP;
- T_0 is the initial retransmit timeout in seconds;
- b is the number of packets acknowledged by a received ACK; and
- p is the packet loss rate.

Tanenbaum [8] summarises this approximation as “*the throughput goes up as the inverse square-root*

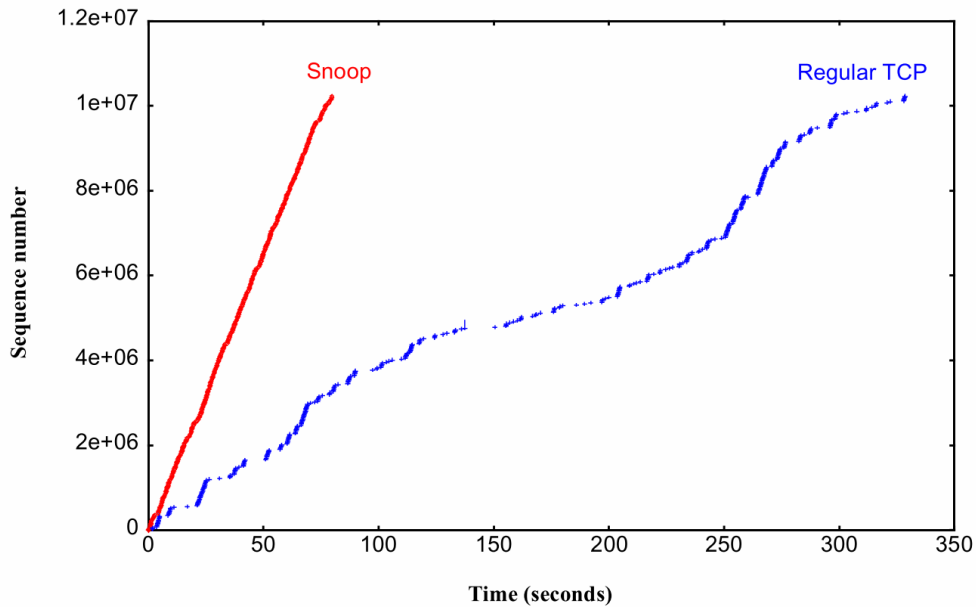


Figure 2.7 Snoop vs. Regular TCP comparison: sequence numbers of received frames against time [60].

of the packet loss rate”. In practical terms this means that the throughput with a packet loss of 10% is not simply ten times slower than the throughput with a packet loss of 1%; instead, the network effectively ceases to function at all. Furthermore, frame loss rates of 10% and greater are common for 802.11 wireless networks which can cause TCP transmissions to suffer.

Improvements in the network protocols are needed to address this problem. One such solution is referred to as loss masking or the fast-retransmit approach. The approach ensures that retransmissions happen in a timescale of microseconds to milliseconds, whereas congestion control is applied in the timescale of milliseconds to seconds. In this case, retransmissions can be triggered and completed before the transport protocol can incorrectly detect a congested network. Only if the retransmissions continue for a long enough time is congestion control applied. This enables transport protocols to work well in most wireless networking environments, with one notable exception: links with a long *round-trip-time* (RTT), such as satellite communication [8], [60].

A second solution to this problem is the Snoop protocol [60]; an adaptation of the TCP protocol with optimisations specifically designed for wireless networks, including:

- caching of unacknowledged TCP data packets;
- identifying packet loss using duplicate acknowledgements; and
- using local retransmissions of the cached packets once packet loss is detected.

Figure 2.7 shows how these optimisations deliver significantly improved performance when compared to unmodified TCP and is an example of how protocol optimisations can improve the performance of wireless networks. Unfortunately, optimisations such as those implemented in Snoop cannot simply be included in a future update of TCP to suit wireless networks.

A protocol that is so ubiquitous can be very difficult to update. Many devices required for the functioning of extended networks will not be updated and will have to be replaced. This is referred to as the ossification of a protocol, a biological term which refers to “*a tendency toward or state of being molded into a rigid, conventional, sterile, or unimaginative condition*” [61]. TCP is one of the best examples of protocol ossification. Papastergiou et al. [62] surveyed many proposed solutions to the ossification of the Internet’s transport layer and concluded that a way forward is a transport layer framework rather than the handful of core protocols used today. The requirements for a future evolutionary transport framework are presented in Figure A.2 on page 124.

2.2.5 Network performance evaluation

In this section, the four metrics used to quantify and evaluate network performance are presented. These four metrics are known as the Quality of Service (QoS) metrics and are presented with formal mathematical definitions. In the evaluation of the artefact, presented in Chapter 4, these metrics are measured using software utilities.

Bandwidth

Bandwidth refers to “*the ability to move a volume of data over a unit of time*” [8]. The upper bound for bandwidth over a noisy transmission medium is calculated using Shannon’s theorem for channel capacity [63], [64]:

$$C = B \log_2 \left(1 + \frac{S}{N} \right) \quad (2.2)$$

Where:

- C is the maximum channel capacity in bits per second;
- B is the bandwidth of the channel in Hz;
- S is the signal power in dB; and
- N is the noise power in dB.

Latency

Latency is defined as “*the time interval from starting an application data transmission of a message at the producer until a successful reception of the message at the consumer*” [65]. A single sample of network latency can be expressed mathematically as:

$$T_{lat} = t_i^{Rx} - t_i^{Tx} \quad (2.3)$$

Where:

- i is the packet number;
- t is the time stamp;
- Rx indicates packet reception; and

Tx indicates packet transmission.

Typically, a single sample of latency is not considered, but rather the mean value of multiple latency samples. As the calculation uses timestamps from a separate sender and receiver, this calculation of latency requires the sender's and receiver's time to be synchronised.

Jitter

Jitter is commonly expressed as the standard deviation σ of a given sample, but the validity of this expression requires the sample of latency values to be normally distributed. Jitter is also defined as the difference between the P95 and P5 percentiles of the sample. Jitter is defined as [65]:

$$J_{lat} = Max(T_{lat}) - Min(T_{lat}) \quad (2.4)$$

Where:

J_{lat} is the variation in latency of the sample;
 $Max(T_{lat})$ is the P95 percentile of the latency T_{lat} ; and
 $Min(T_{lat})$ is the P5 percentile of the latency T_{lat} .

In practice, jitter is calculated for multiple sequential latency samples and the mean of these samples is used.

Packet Loss

The packet loss rate is expressed as the ratio of successfully transmitted traffic flows and all transmitted traffic flows. Assuming a set of cyclic real-time traffic flows $F = f_1, f_2, \dots, f_n$ from the application, each transmitted x times, the packet loss rate λ_{PL_i} for a given flow $1 < i < n, n \in N$ can be expressed mathematically as [65]:

$$\lambda_{PL_i} = \frac{x_{f_i}Tx - x_{f_i}Rx}{x_{f_i}Tx} \quad (2.5)$$

Where:

λ_{PL_i} is the packet loss rate of the sample;
 $x_{f_i}Tx$ is the number of packets transmitted in the sample; and
 $x_{f_i}Rx$ is the number of packets successfully received in the sample.

These four metrics, expressed formally in Equations 2.2 to 2.5 on pages 28–29, represent the core networking characteristics used to evaluate networks quantitatively. Secondary characteristics might include network security, scalability, energy efficiency and cost of deployment, but are not considered in this research.

2.3 Distributed Computing

A distributed computing system is defined as “a system where cooperating nodes communicate and coordinate their actions by passing messages” [10]. From the user’s perspective, it is “a collection of independent computers that appears to its users as a single coherent system” [42].

Michael J. Flynn proposed a taxonomy of “very high-speed” computer architectures in 1966 [66]. His work has remained remarkably relevant as a tool in the design of modern processors [67]. In Flynn’s taxonomy, computing systems can be classified in terms of [68]:

- Instruction Streams (to be executed on a data stream); and
- Data Streams (to be operated on by an instruction stream).

As shown in Table 2.7, this gives four distinct classifications of computing systems. An extension of Flynn’s taxonomy is the User-Application (U-A) taxonomy [69]. It adds how computer systems are used to how they are internally constructed by prepending the number of users (SU/MU) and the number of applications (SA/MA) to the degree of parallelism (provided by Flynn’s taxonomy). For example, a modern smartphone would classify as Single User, Multiple Application - Multiple Instruction, Multiple Data stream (SUMA-MIMD) as the device runs multiple applications concurrently on a multiprocessor but is typically only used by a single user. In contrast, an example of an MUSA-MIMD architecture would be a web server that hosts a web application: it is a single application that is simultaneously used by many users, but its data is exchanged in a client-server model (e.g. a REST API).

	Single instruction stream	Multiple instruction streams
Single data stream	SISD	MISD
Multiple data streams	SIMD	MIMD

Table 2.7 Flynn’s taxonomy [68].

Parallel computers, including modern personal computers and distributed computing systems, are either classified as SIMD or MIMD. They are classified as SIMD when there is only one control unit, and all processors in the machine execute the same instruction on all data streams at the same time. In MIMD systems each processor has its own control unit and can execute different instructions on different data streams [70]. Within this architecture, systems can be classified into two categories relating to the way their inter-processor coordination is implemented:

- Shared memory; and
- Message passing.

Figure 2.8 shows an MIMD architecture that uses shared memory. Systems implemented in this way coordinate their processing by using a single global memory pool used by all processors. These systems are also referred to as SMP (symmetric multiprocessor) systems because access to the memory (M) is divided equally among processors (P) [70].

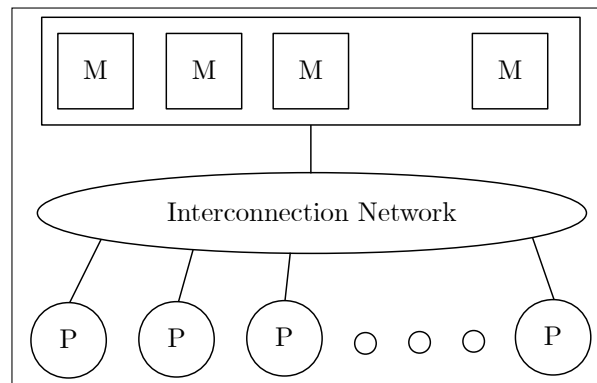


Figure 2.8 MIMD architecture using shared memory [70].

Figure 2.9 shows an MIMD architecture that uses a message-passing system to coordinate processing. In this implementation, processors (P) may need to process data stored in memory (M) allocated to another node. Message passing is used to transfer the required data between the memory of the node that has it and the memory of the node that needs it. While this implementation requires more effort from the programmer, it has become apparent that this distributed memory implementation scales efficiently [70].

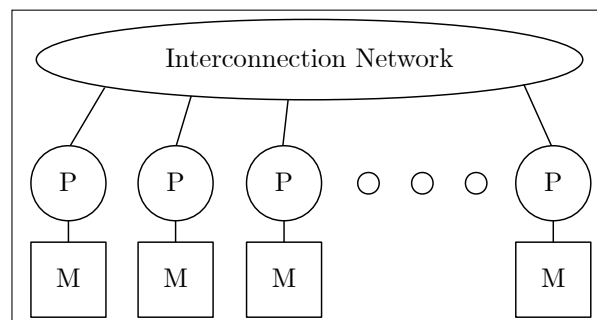


Figure 2.9 MIMD architecture using message passing [70].

2.3.1 Classifying distributed systems

Within the MIMD distributed computing architecture, several distinct classes of distributed computing implementations can be found. In this section, the following three distinct implementations described by Tanenbaum and Van Steen are presented [42]:

- Distributed computing systems;

- Distributed information systems; and
- Distributed pervasive systems.

Distributed computing systems

Examples of this class include cluster and grid computing systems. Cluster computing systems aggregate the processing power of many homogeneous computers using a high-speed, dedicated network and are used to solve computationally intensive tasks running in parallel [71]. Grid computing systems are very large-scale networks of computers spanning multiple organisations and administrative domains [72] that agree to share their computational resources. Grids are similar in application to a cluster computing system, with some key differences, including the heterogeneity of the computation nodes, the scale of implementation and no reliance on the availability of individual nodes.

Distributed information systems

Examples of this class include transaction processing and enterprise application systems. Transaction processing systems are large data-centric systems that store information across multiple machines to increase availability, performance and disaster recovery [42]. Enterprise application systems enable a large organisation to create other applications that integrate data across departments for use by employees and clients. Although these systems often use a server-client architecture, there is a move toward event-driven, streaming architectures in this sector [73].

Distributed pervasive systems

Examples of this class include home systems, electronic healthcare systems and sensor networks [42]. Home systems refer to a network consisting of personal computers, televisions, smartphones, wearables, light controllers and other consumer electronics combined into a single distributed system. Electronic healthcare systems include multiple devices/sensors that a person wears to form a body area network (BAN) to monitor the body, and possibly perform some action, such as administering a drug or logging sensor results. Sensor networks are information collection networks consisting of many small nodes, each equipped with sensors. The nodes typically communicate through a wireless network and are used for monitoring applications.

2.3.2 Distributed and parallel computing

Parallelism in a computing context refers to the use of two or more processors (often referred to as processing cores in systems with multiple processors on a single processor die) to solve a single problem [74]. Parallel computing has become the norm, as the vast majority of traditional computers and other computational devices used today are capable of running multiple processes at once.

Table A.1 on page 124 shows a range of devices, including networking equipment, IoT devices, smartphones, laptops, workstation desktops and GPUs that are all equipped with multi-core

processors capable of parallel processing. It is important to note that these processors are designed for use in different applications and implement significantly different architectures. The CUDA cores in the Nvidia TU102 (Quadro RTX 6000) are not directly comparable to the ARM or x86/AMD64 cores used in general purpose processors. The purpose of this table is to show that modern processors, whether used in a \$35 Raspberry Pi or a \$6,300 Nvidia Quadro GPU, all leverage parallel processing for their intended application.

According to Flynn [66], the term parallelism is insufficient when considering technical details on execution. Parallelism is “*both ubiquitous and ambiguous*” and therefore Flynn proposes two preferable working terms: instruction streams and data streams. These terms are the basis for Flynn’s taxonomy presented in Table 2.7 on page 30. From these terms, two approaches to parallelism emerge, presented in Table 2.8.

Approach	Description
Task parallelism	The programmer divides computation work into distinct tasks which are mapped to the number of physical threads scheduled by the operating system. Libraries, such as Portable Operating System Interface (POSIX) and Intel’s Thread Building Blocks (TBB), can be used by programmers to leverage task parallelism.
Data parallelism	The programmer defines tasks that perform the same operation on different pieces of data. The Hierarchically Tiled Array (HTA) library facilitates data-parallel execution programs and uses tiling to define locality and parallelism.

Table 2.8 Approaches to parallelism [75].

Pandey et al. [76] present examples of real-world concurrent workflows from the domains healthcare, distributed robotics and computer vision. Each of the workflows presented in Figures A.3 to 2.10 on pages 126–34 use the following notation for sub-tasks:

$$\alpha_n^k$$

Where:

- α is the sub-task of the overall goal (denoted by k or l);
- k represents the task number (vertical sequence) of the workflow; and
- n represents the data-source number (horizontal sequence) of the workflow.

Figure A.3 on page 126 presents the healthcare example with two goals: stress detection with its sub-tasks denoted k and hypoxia detection with its sub-tasks denoted l . The healthcare example is implemented using a purely task-parallel algorithm. Figure A.4 on page 126 presents a workflow from the computer vision domain with two goals: canny-edge detection and scale invariant feature transform (SIFT). As in the previous example, the sub-tasks of these two goals are denoted with

k and l . In contrast to the healthcare example, this application is implemented using a purely data-parallel algorithm.

The final example is from the distributed robotics domain and is presented in Figure 2.10. It has two goals: oceanographic phenomena tracking and intruder detection with their sub-tasks denoted k and l , respectively. This implementation includes a mix of both task parallelism and data parallelism.

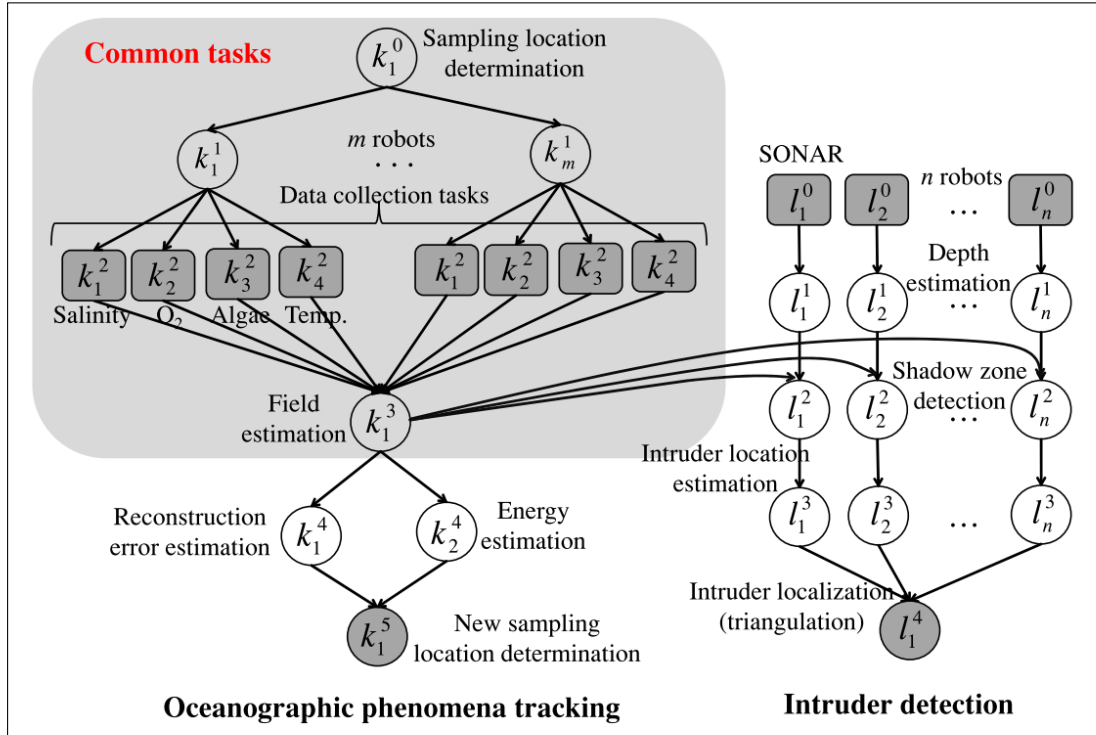


Figure 2.10 Workflow from the distributed robotics domain [76]. The implementation is an example of a workflow with a mix of task parallelism and data parallelism.

The application of task parallelism and data parallelism or the combination of the two is used in many computer applications. Modern computers include processors that can run these applications while using multiple cores for performance gains (see Table A.1 on page 124) rather than using a single core running at higher frequencies. Software compilers and application code need to be optimised to use these cores effectively, as application performance will depend on effective parallelisation across the available cores. When the typical modern computer with a single processor die that includes multiple cores does not satisfy the performance requirements of an application, the processors of multiple distinct machines can be leveraged.

Three widely recognised distributed computing solutions include clusters, grids and cloud. Figure 2.11 on the following page shows the relationship between different distributed computing solutions in terms of scale and whether the technology better suits application or service-oriented operation. A comprehensive table of comparisons between these three solutions is presented in Table A.2 on page 125. The categories of comparison that are most relevant to this research include loose coupling

and reliability. Cloud computing are considered superior to clusters in both of these categories. The technologies and approaches associated with cloud computing are therefore better suited to the goal of research than the classic cluster computing approach.

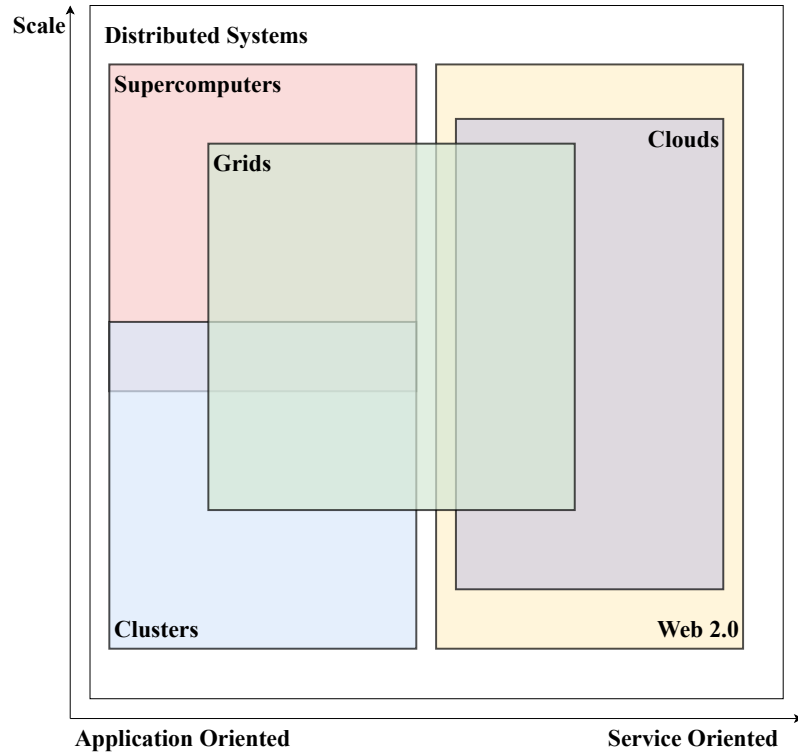


Figure 2.11 Relationship between distributed computing solutions [77].

2.3.2.1 Clusters

A cluster is defined as “a collection of distributed computers which are interconnected among themselves using high-speed networks, such as Gigabit Ethernet, SCI, Myrinet and Infiniband” [71]. Computers in a cluster are often referred to as nodes and dedicate all of their processing towards a task chosen by a controller. Clusters are built from commercial-off-the-shelf (COTS) components instead of custom-designed hardware as seen in application-specific supercomputers. Nodes communicate over a high-speed network using software libraries, such as message-passing interfaces (MPI).

Examples of popular open-source implementations of the MPI standard, set by the MPI Forum [78], include OpenMPI [79] and MPICH [80]. The MPICH implementation includes library, compiler and runtime components [81]. When developing an application to be run in the MPI environment, the developer will reference the library component in their C application code. This library provides functions to be used during execution such as *MPI_Send* and *MPI_Receive*. The developer will then compile their code using the appropriate MPI compiler to create an executable binary intended to be run using the compatible MPI runtime.

An architectural representation of a distributed application running in an MPI environment is shown

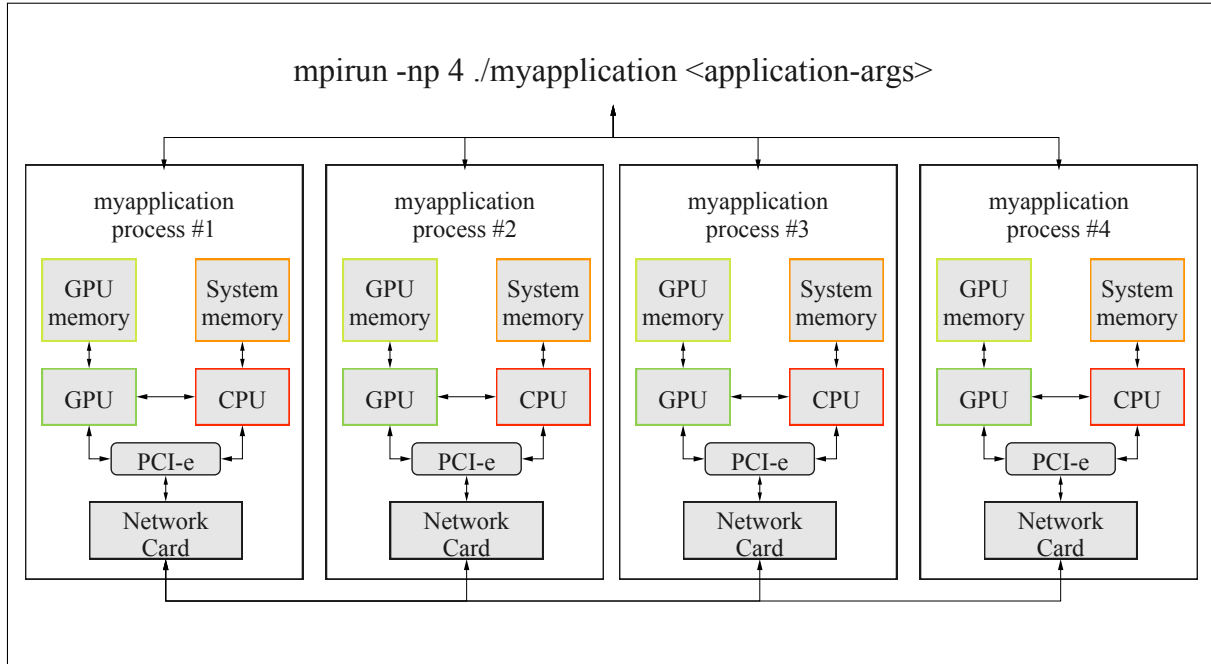


Figure 2.12 MPI application architecture. Figure adapted from [81].

in Figure 2.12. The MPI runtime creates a specified number of processes spread across the available number of nodes. The `-np 4` argument to the MPI runtime specifies that this application should be run using four processes.

2.3.2.2 Grids

Foster [82] developed a three point checklist for what represents a grid by combining the essence of several definitions. A grid is a system that:

1. coordinates resources which are not subject to centralised control;
2. uses standard, open, general-purpose protocols and interfaces; and
3. delivers nontrivial qualities of service.

Grids distribute computational tasks across many systems, similar to the aforementioned cluster computers. However, grid systems are typically distributed across a much larger scale and might contain multiple clusters [83]. Grid systems can span multiple organisations, cities, countries and even continents, but are not necessarily more or less powerful than dedicated clusters. Well-known examples of global grid systems include BOINC [84] and Folding@Home [85], which use grid-computing techniques applied in a volunteer-computing arrangement.

In volunteer-computing arrangements, the systems participating volunteer a portion (e.g. when idle) or all of their computational resources to support some cause, e.g. medical research. Participation is often incentivised using competition and gamification principles, where users compete to donate more computation power than their local and international competition, judged by a built-in ranking

system².

As a consequence of the voluntary agreement, the system cannot rely on any node's continued availability. This is the most relevant difference between grid computing and high-performance computing clusters for the application area: grid systems need to cope well with an ever-changing pool of computing resources.

2.3.2.3 Cloud computing

Cloud computing is defined as “*a type of parallel and distributed system consisting of a collection of interconnected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on a service-level agreement*” [71]. Cloud computing is presented as a reference of how most modern distributed computing systems are built, but is not investigated in-depth as it is not relevant to building a localised wireless distributed computing platform.

2.3.3 Wireless distributed computing

Wireless distributed computing (WDC) is a research field at the intersection between distributed computing and wireless networking. In WDC, processing workloads are distributed over wireless connections between participating nodes. These nodes process workloads assigned to them and return results to other nodes that request it over the wireless network [11]. The goals of wireless distributed computing include reducing requirements of a system in terms of its per-node power, network power, energy and processing requirements [11].

WDC enables the deployment of distributed systems in dynamic environments and is made possible due to technological advances, such as fault-tolerant computing, distributed computing and software-defined radio. Datla et al. [11] identified four classes of WDC applications, including:

1. Real-time data capture, processing and dissemination;
2. Complex communication waveforms;
3. Information sharing; and
4. Robust process control.

Of these classes, the first and fourth classes are the most relevant to this research. Consider the Intel drone swarm presentation presented in the introductory paragraph of Chapter 1. In order for the drone demonstration to function, the system needed to perform time-sensitive collection and processing of location-based data. The demonstration also represented a process control problem, where the process was the changing formation of individual drones over time.

Figure 2.13 on the following page shows an example of a robot swarm application in which these concepts could be applied in a complementary fashion. Swarm robotics is investigated in more detail in Section 2.5 and this section is focused only on the computation aspect. Each of the drones

²https://setiathome.berkeley.edu/top_users.php

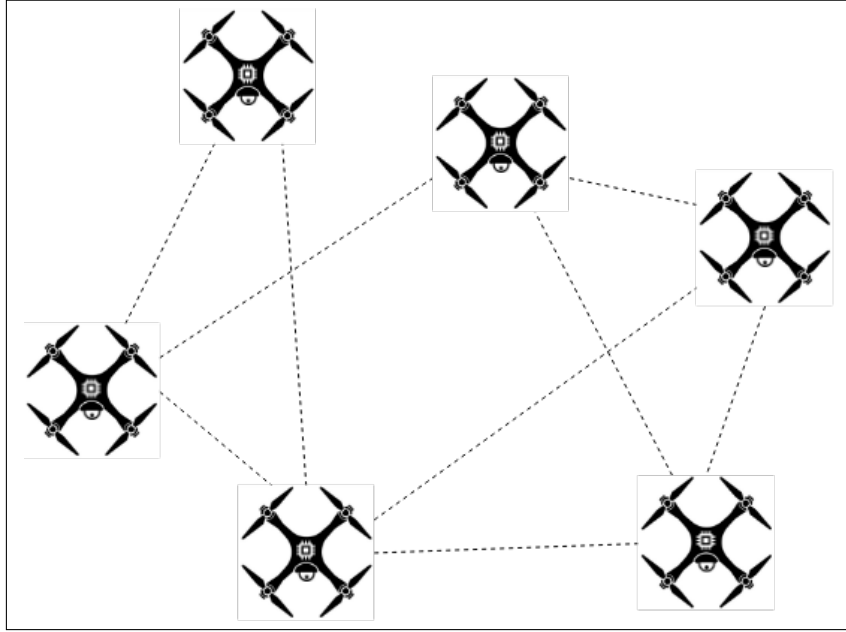


Figure 2.13 Swarm of drones in which each drone has local sensing, computation and networking capabilities.

in the figure has localised sensors, such as GPS, an ultrasonic height sensor and a video camera. Each drone also has a relatively small amount of processing power and is connected via a mesh network. The real-time processing of location data produced by the drone swarm's collection of sensors could be accomplished in several ways:

- *Local centralised computing* - processing performed on a dedicated machine nearby (e.g. a nearby machine). This is the approach used in the Intel demonstration;
- *Remote centralised computing* - processing offloaded to a remote server or cloud infrastructure;
- *Local distributed computing* - processing using the collective power of each drone's processing power; and
- *Remote distributed computing* - processing offloaded to a remote distributed computing environment.

The first choice is between local and remote processing, both of which have advantages and disadvantages. Local processing does not require an internet connection in order for the system to function and can, in some cases, provide quicker results. The processing power is limited to that of the machine available, whereas remote processing can access nearly unlimited computing power via a public cloud provider. A combination of local and remote processing could also be used. Such an approach is often referred to as fog computing, since some computing is performed in the cloud while other computing tasks are completed locally - the fog is essentially at the edge of the cloud [86]. An example in which fog computing is used is Wireless Sensor and Actor Networks (WSAN). These systems share some similarities with the model presented in Figure 2.13, as they contain many nodes with low bandwidth, a small amount of processing power and limited memory

resources. The nodes will continuously provide sensor data from the environment and can optionally influence it. Nodes in WSN deployments often operate on battery power and to conserve energy, perform only time-critical computation locally and offload less time-critical computation to the cloud [27].

The second choice is between centralised and distributed computing, which also have advantages and disadvantages. Centralised processing represents the simplest approach, but introduces a single point of failure in the system. Distributed computing eliminates the single point of failure, but adds complexity. Intel used local centralised computing for their opening drone show at the 2018 Winter Olympic games [1], as mentioned in the introductory paragraph of Chapter 1. This approach proved successful in this case, but an alternative approach merits investigation.

2.4 System Architecture

Architecture is defined as “*the art or science of building*” and systems architecture is defined as “*the manner in which the components of a computer or computer system are organised and integrated*” [87]. Douglas [88] defines the process of architecture as “*the set of strategic design decisions that affect most or all of the system*”. The end result of this process is a system architecture intended to solve a specified problem, often represented in logical diagrams. A comparison between two system architectures is frequently found in literature: the monolith and microservices. It is also clear from recent literature that there is a shift from the traditional monolithic architecture towards the more modern microservice architecture [89]–[93]. In this section, these two architectures are introduced.

2.4.1 Monoliths

A monolithic application (also referred to as a single executable artefact) is defined by Dragoni et al. [94] as a “*software application whose modules cannot be executed independently*”. Monoliths have limited modularity, since their modules typically rely on the sharing of resources, such as memory, databases and files, on a single machine. These modules can exist independently of each other, but are of limited utility until they are bound together by the executable. Efforts toward making monolithic applications more suitable for distributed application environments (including the use of network objects) still leave multiple challenges, including [93]:

- *Maintenance difficulty* - Monoliths are inherently complex and therefore, enhancements and debugging require plenty of time;
- *Dependency hell* - With so many dependencies supporting a single system, keeping them up to date becomes very complex due to incompatibilities of library versions, leading to inconsistent or non-functional systems;
- *Increased downtime and slower development* - Minor changes to a single component in the monolith requires the recompilation and deployment of the whole application, which takes a considerable amount of time. This slows down development, testing and maintenance;
- *Sub-optimal deployment* - Developers need to choose a “*one size fits all*” configuration.

Some modules might be memory-intensive while others require more processing resources and choosing a single solution able to support all modules results in overly expensive or sub-optimal deployments;

- *Limited scalability* - Typical scaling in response to requests involves adding more instances of your application on more servers (horizontal scaling) and balancing the load between them. With a monolithic architecture, the system needs to be scaled as a whole instead of just scaling the subsystems that are under stress; and
- *Technology lock-in* - Tightly coupled frameworks and languages that make up the system cannot be easily substituted when a better alternative becomes available.

As an example, Figure A.5 on page 127 shows an architectural model of the monolithic FX Core bank system used at a major bank in Denmark. This model visualises some of the complexity of a monolith and the tightly coupled nature of the components. The aforementioned challenges are the impetus behind the transition towards highly scalable microservice architectures, discussed in the next section.

2.4.2 Microservices

A microservice is defined as a “*cohesive, independent process interacting via messages*”. Modularity is an important component of a microservice; for example, a service that performs a calculation on a data set is not also responsible for plotting/displaying the result. The microservice architecture is characterised by a “*distributed application where all its modules are microservices*” [94]. The modularity of the architecture simplifies development, maintenance and testing of each individual microservice, since its dependencies and configuration can be managed without regard for the services that interface with it via messages. Microservices are not linked to any development paradigm, language or technology, but instead provide a guideline to the partitioning of components within a system and address the challenges of monoliths in the following ways [94]:

- *Limited individual functionality* - Each microservice has a small codebase focused only on providing the correct response to incoming requests. This makes them significantly easier to test and maintain;
- *Continuous integration* - A gradual transition to a newer version of the microservice is possible. When upgrading dependencies, it is possible to verify correct behaviour in a limited scope. A newer version can also be run concurrently with an older version;
- *Optimised deployment* - Upgrading to newer versions of each microservice does not necessarily require system downtime, since only that microservice needs to be redeployed;
- *Containerisation* - Microservices are easily deployed as a set of containers. This gives developers a higher degree of freedom to deploy the microservice in an extremely scalable orchestrated environment such as Docker Swarm or Kubernetes. Containerisation and container orchestration is presented in Section 2.4.4;
- *Dynamically scalable* - Scaling a system implemented as a microservice architecture implies

scaling only the components that need to be scaled based on load, instead of the entire system; and

- *Low commitment to specific technology* - The only constraint present in a microservice architecture is the network facilitating communication between each service. Beyond that, developers are free to leverage various technologies (language, framework, etc.) to best support a microservice's intended functionality.

Figure A.6 on page 128 shows the replacement FX Core banking system, implemented using a microservice architecture. This implementation allowed them to leverage modern architecture practices for increased robustness and availability [93]:

- *Containerisation* - A Docker Swarm Cluster runs all services, allowing the entire architecture to be deployed with a single command. It also allowed developers to run a development environment on their equipment that is practically identical to the production environment;
- *CICD pipeline* - A continuous integration and continuous deployment (CICD) pipeline was leveraged to achieve completely automated infrastructural tasks, including rolling updates;
- *Orchestration* - With containerised services, Docker Swarm was used as an orchestrator to manage automated replication of service containers;
- *Clustering* - Five virtual hosts located in three data centres run the Docker Swarm supporting the architecture, providing massive redundancy and availability benefits. Docker Swarm also provides overlay networking, allowing containerised services to communicate without knowledge of the complex network infrastructure that supports it;
- *Load Balancing* - Docker Swarm's ingress load-balancer functionality can direct incoming requests to separate replicas of the same container to balance the load between them. If a container fails, the load balancer will not send requests to it and wait for its replacement to be deployed before directing requests to it. This approach can prevent system faults in a container from presenting as system errors to the users; and
- *Service Discovery* - Docker Swarm's network tooling ensures that deployed services are assigned hostnames that can be resolved by other containers in the Swarm. One container on the same virtual overlay network can communicate with another container without needing to know its internal IP address.

In the case study of the FX Core banking system, Dragoni et al. [93] concluded that re-engineering the system as a microservice architecture provided reduced complexity, lower coupling, higher cohesion and simplified integration. Microservices have proved very successful in web-scale systems, such as Netflix, which was one of the earliest adopters of the monolith to microservice migration process. Netflix serves high-fidelity video content to millions of users at an affordable price. This was made possible with a robust, auto-scaling, highly distributed system with automated deployment supported by a microservice architecture.

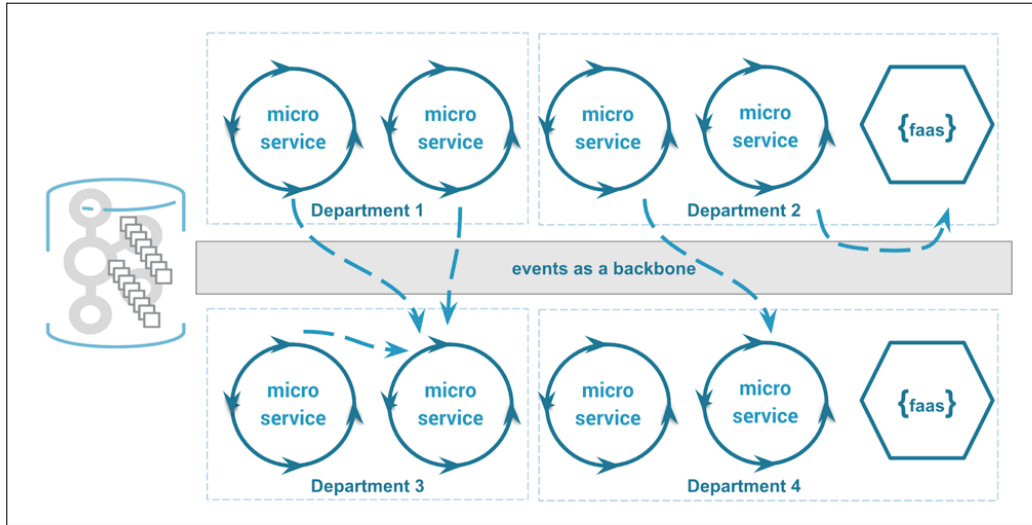


Figure 2.14 Using stream processor microservices to build organisational processes as the flow of atomic events [95].

2.4.3 Event-driven architecture

Event-driven architecture refers to a design thinking and programming model that fundamentally views the world as a set of atomic events. Examples of events in a system might include message received, stock price updated or client logged in. As Avery [95] puts it, *“the key realisation for the adoption of event-first thinking is that an event represents a fact, something happened; it is immutable and therefore changes how we think about our domain model”*. From an organisational view, events can be used as a backbone, shown in Figure 2.14.

Programming models have evolved as computational problems have evolved over time [96]. The use of distributed objects, service-oriented architecture, enterprise service bus and event-driven architecture have built on the lessons learnt from their predecessors. Their creation was a reaction to existing solutions not being capable of solving the design problems at hand. As the Reactive Manifesto [97] notes, *“today’s demands are simply not met by yesterday’s software architectures”*. The recommendation of the Reactive Manifesto is for systems to be:

- *responsive* - respond in a timely manner if at all possible;
- *resilient* - stay responsive in the face of failure, using replication, containment, isolation and delegation;
- *elastic* - stay responsive under varying workloads by increasing or decreasing the computational resources that support them based on their utilisation; and
- *message driven*: rely on asynchronous message passing to ensure loose coupling and isolation.

In contrast to the traditional request-driven architecture that is often found in client-server applications, event-driven streaming architectures make use of an event distribution layer as shown in Figure 2.15 on the following page. A key difference is that the distribution layer keeps long-

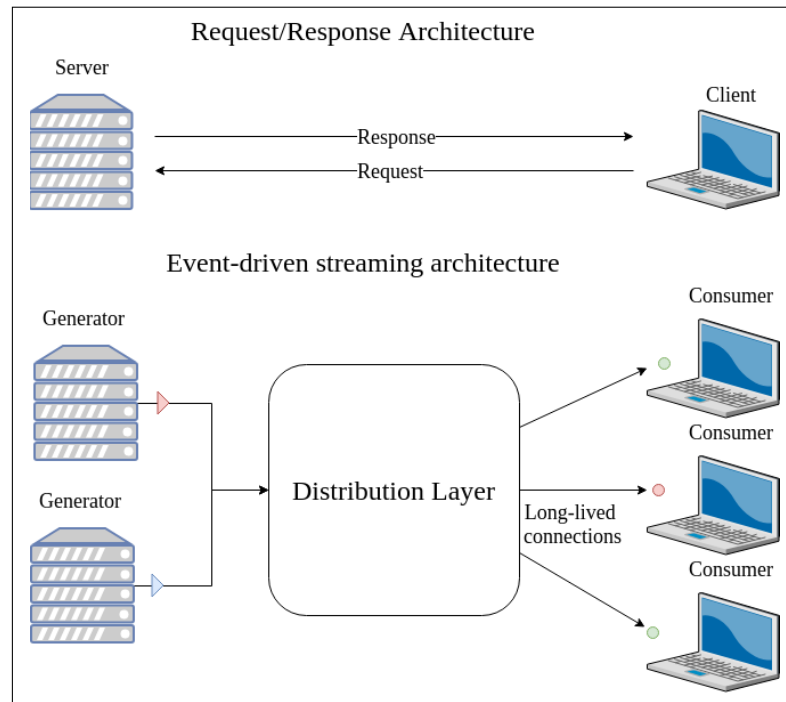


Figure 2.15 Comparison of request-driven architecture to event-driven streaming architecture [73].

lived connections with clients. This approach facilitates responsiveness, and since the clients use asynchronous execution, processor time is only used when needed.

During a presentation at DevConf South Africa 2019³ entitled “*RESTful must die! Challenges building realtime AI at scale.*”, Wells [98] presents an evolutionary perspective on systems architecture. In this perspective, Wells divides systems architecture into four generalised versions:

1. A centralised system with a database for storage. The application is tightly coupled and therefore does not scale well;
2. A distributed system with a front-end and back-end. The front-end is concerned with the presentation of information while the back-end is concerned with applying business logic to information stored in a database;
3. A distributed system with the back-end split into multiple interdependent microservices (refer to Section 2.4.2); and
4. A distributed system where the database has been replaced with distributed event stores and the traditional front-end and back-end are implemented using stream processors.

The four versions are represented in Figures 2.16 to 2.17 on pages 44–45. Each of the versions improved upon the limitations presented by their predecessor and each design was better suited to meet the demands of the applications that implemented them.

The fourth version, presented in Figure 2.17 on page 45, represents Wells’ suggestion for next

³<https://www.devconf.co.za>

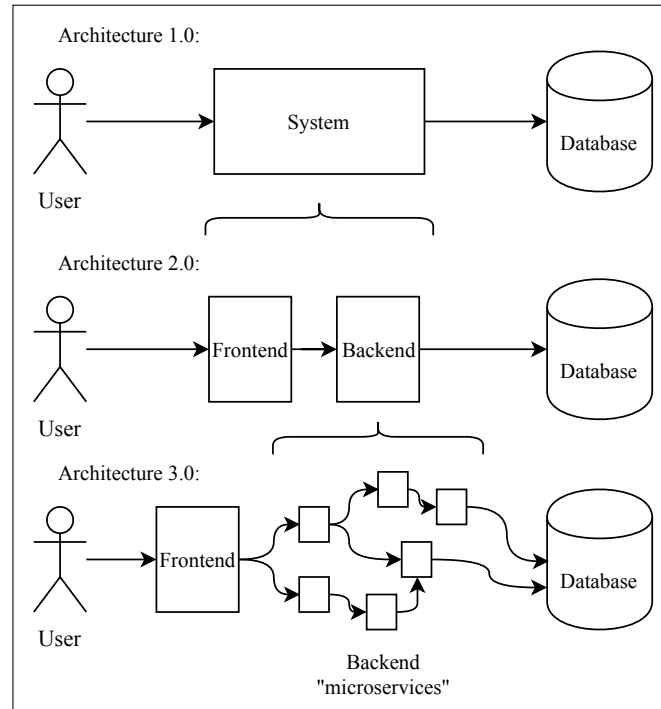


Figure 2.16 Evolutionary perspective of system architectures [98].

evolution of systems architecture. In his view, future systems could be designed around the concept of event-streaming, presented in Section 2.4.3. This view seems to be shared by Avery [95] in his article titled “*Journey to event driven – Part 1: Why event-first thinking changes everything*”.

Consider the transitions between versions of systems architecture as presented in Figure 2.16. The first transition implemented service-oriented architecture (SOA) to decouple the system into two parts: a front-end and back-end. This decoupling allows a separation of concerns and allowed systems to operate beyond the one-to-one relationship between service and presentation layers to form many-to-many relationships instead. The next step towards application scale came with the adoption of microservices (refer to Section 2.4.2).

Netflix⁴ represents an excellent early example of how the shift to microservice architecture enabled the streaming of high-definition video to millions of clients at a low cost while increasing fault tolerance through the use of stateless microservices [99]. Each service is small, performs a specific functionality and can be scaled dynamically to enable the application to deal with unpredictable load.

In Wells’ presentation, he notes that “*breaking up the stateless parts of our system didn’t get us all the way there; we also needed to break up the storage layers*”. The centralised state store is the next barrier to overcome and the final transition between version 3.0 and 4.0 shows this change. The database management systems included in versions 1.0 through 3.0 have been replaced by distributed

⁴<https://www.netflix.com>

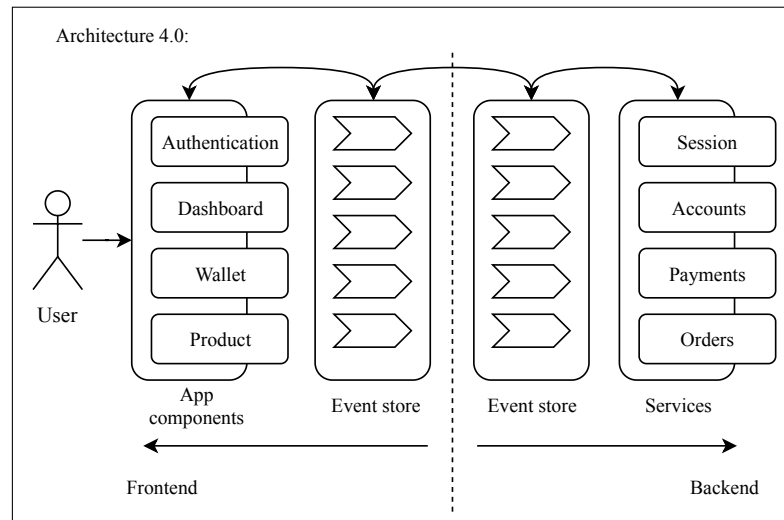


Figure 2.17 Potential future of system architectures [98].

event stores. By treating everything in the system as an event, the hierarchy of components in the system is implicitly flattened. The motivation for such a change is application scale; centralised databases that serve requests to many services became the bottleneck once microservices enabled application scale.

Avery notes that reactive systems “are significantly more tolerant of failure and when failure does occur, they meet it with elegance rather than disaster”. The aim of the distributed computing platform presented in Chapter 3 is to be more fault tolerant. Towards this aim, the use of event-driven architecture is practically investigated in Chapter 3.

2.4.4 Containerisation

In this section, the practice of containerisation is introduced. Containerisation is a lightweight alternative to full machine virtualisation. A key distinction between virtual machines and containers is that containers are virtual runtime environments and do not run full operating systems as in the case of virtual machines [100]. A container engine is responsible for creating and managing the virtual environment in which these containers are designed to run.

Figure 2.18 on the following page presents a comparison between multi-core processing, virtualisation and containerisation. On the left, is the standard multi-core processing paradigm as a reference. In this paradigm, a computer has a processor with multiple processing cores running a single operating system. The operating system enables software processes to perform their intended processing using system calls.

In the middle, is the virtualisation approach, where a hypervisor is introduced between the operating system and the hardware. Each virtual machine still runs a full operating system which in turn supports software processes. Two distinct types of hypervisors are available, simply referred to as

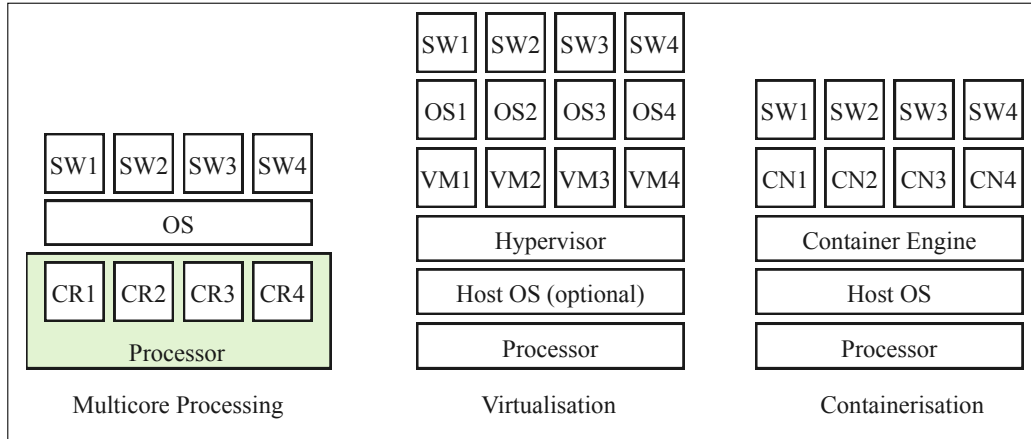


Figure 2.18 Comparison between Multi-core Processing, Virtualisation and Containerisation [100]. *CR*: Processing core, *OS*: Operating System, *SW*: Software process, *VM*: Virtual Machine, *CN*: Container.

Type 1 and Type 2 [101]. Type 1 hypervisors are also referred to as bare-metal hypervisors, as they do not require a host operating system - the operating systems in the virtual machine get direct access to the hardware. Examples of such hypervisors include *VMWare ESXi*⁵ and *Microsoft Hyper-V Server*⁶. In contrast, a Type 2 hypervisor is installed alongside an existing operating system and is often referred to as a hosted hypervisor. The hypervisor software relies on the host operating system to manage calls to system resources, including CPU, memory, storage and networking. One such example is *VMWare Workstation*⁷.

On the right, is the containerisation approach. Docker, the most popular containerisation solution by a substantial margin [102], describes containers as “*standardised units of software*” [103]. Containers package application code (or binaries), required runtime software, system tools, libraries and application settings into a single container image that is highly portable between computing environments. A direct comparison between virtual machines and containers is presented in Table 2.9 on the following page, but it is also common practice to use containers and virtual machines together [103]. There are several containerisation solutions available, often with a distinct container image format and a container engine component. As of early 2019, notable containerisation solutions include [102]: *Docker*⁸, *CoreOS rkt*⁹, *LXC/LXD*¹⁰ and *Mesos*¹¹.

Unlike virtual machines, the images of containers are quite minimalist. A goal when developing with containers is to get the image as small as possible, containing just enough software to support the application. The Alpine Linux project¹² is a Linux distribution dedicated to providing a small,

⁵<https://www.vmware.com/products/esxi-and-esx.html>

⁶<https://www.microsoft.com/en-us/evalcenter/evaluate-hyper-v-server-2019>

⁷<https://www.vmware.com/products/workstation-pro.html>

⁸<https://www.docker.com/>

⁹<https://coreos.com/rkt/>

¹⁰<https://linuxcontainers.org/>

¹¹<http://mesos.apache.org/>

¹²<https://alpinelinux.org/about/>

	Virtual Machines	Containers
Level of virtualisation	Hardware level	Operating system level
Weight	Heavy	Light
Provisioning	Slow	Real-time and scalable
Performance	Limited	Native
Isolation	Full, more secure	Process only, less secure

Table 2.9 A direct comparison between virtual machines and containers [104].

secure and simple base image for containers. It requires only 8 MB of storage and provides enough functionality to support very complex applications.

Containers are meant to make a single application extremely portable and typically have a single application process running inside them. To support large applications with multiple components, many containers are orchestrated to work together. As an example, one container might run a .NET Core web-service process that exposes data stored in a MySQL database running in a separate container. As the number of components of applications increases, the deployment complexity increases with it. Container orchestration “*groups the containers that make up the application into logical units for easy management and discovery*” [105].

Container orchestrators, such as the industry-leading *Kubernetes*, enable applications to run with high-availability using techniques including replication, presented in Section 2.1.3. By packaging application components into decoupled microservices as presented in Section 2.4, these components can be scaled individually. Orchestrators make it simple to run multiple instances of a service and load-balance requests between them. This greatly increases fault tolerance by decoupling components and replicating components that could represent single points of failure in the application. Popular orchestration solutions include [106]: Docker’s included Swarm mode¹³, Apache’s Marathon¹⁴ and Google’s Kubernetes¹⁵.

2.5 Swarm robotics

The Intel drone swarm demonstration presented in the introductory section of Chapter 1 provides an excellent visual example of swarm robotics and was a significant source of inspiration for this research. Swarm robotics is a research field that identifies and aims to solve many of the challenges facing these robot swarms, often developing computational algorithms that mimic existing natural solutions employed by animal species such as flocks of birds, ant colonies or swarms of bees [107]. Robot swarms are teams of robots that collaboratively pursue a task that cannot be accomplished by any of them individually [108]. The paradigm draws similarities to distributed computing, presented in Section 2.3, in which many less powerful computers work together to process large tasks.

¹³<https://docs.docker.com/engine/swarm/>

¹⁴<https://mesosphere.github.io/marathon/>

¹⁵<https://kubernetes.io/>

By studying social insects, such as ants and termites, it was discovered that their system-level functioning is robust, flexible and scalable, despite the complete lack of a centralised coordination mechanism. Sahin [108] presents these three observable and desirable characteristics as central motivations for swarm robotics. These characteristics emerge as generalised themes that are relevant to platform robustness (Section 2.1), wireless networking (Section 2.2) and system architecture (Section 2.4). Each of these concepts is presented as they relate to robot swarms.

2.5.1 Swarm robustness

Robustness in the context of a robot swarm refers to the ability of the swarm to continue functioning despite individual failures or disturbances in the environment. Robustness in a networking context was discussed in Section 1.1.1. Swarm robustness is characterised by [108]:

- *Redundancy* - If an individual node in the system is lost or malfunctions, the loss can be mitigated by another node;
- *Decentralised coordination* - There is no single point of failure, and destroying any single part of the larger system will not entirely inhibit its operation;
- *Simplicity of the individuals* - Multiple simple, and therefore more reliable, nodes are preferable to a single complex system capable of performing the same task; and
- *Multiplicity of sensing* - The signal to noise ratio is improved by the use of a large number of distributed sensing nodes.

Ant colonies demonstrate all four of these characteristics. Ant colonies certainly display redundancy; killing several or even thousands is often not enough to successfully eradicate the colony. The queen ant might be killed, but large colonies have multiple reproductive queens [109]. Decentralised coordination is also displayed by ants. Using pheromone secretions for ad hoc task recruitment, ants self-organise into groups dedicated to various tasks [110]. Individual ants are very simple in comparison to the colony and each individual ant's localised senses can be leveraged in order to address complex tasks. These characteristics make ants an excellent subject for analysis when trying to improve robust operation in artificial robot swarms.

2.5.2 Swarm flexibility

Swarm flexibility refers to the ability of the swarm to generate modularised solutions to different tasks. This is also effectively demonstrated by ant colonies. Each individual ant must engage in very different tasks, including foraging, prey retrieval and chain formation [108]. Foraging requires ants to act semi-independently in the search for food sources. Partial coordination is done via pheromones so that ants do not forage the same area multiple times. In contrast, prey retrieval requires full coordination: the group of ants needs to generate a pulling force large enough to move the prey to their nest. The collective force needs to be generated simultaneously and in the right directions for the prey to be moved closer to the nest. Some ants might dedicate their force to lifting the prey, while others dedicate theirs to pulling the prey. These small applications of force

spread over the prey result in a coordinated pulling force towards the nest. In chain formation, ants form a physical chain structure with their bodies. The chain extends beyond the reach of an individual ant and can exert considerable pulling forces on objects, such as leaves. Robot swarms should exhibit the flexibility to offer solutions to the task at hand, using a coordination strategy that suits the change in the problem and environment.

2.5.3 Swarm scalability

The final motivation for swarm robotics is scalability, which refers to the ability of the swarm to operate over a wide range of swarm sizes. While the potential effect that the swarm can enact on its environment is limited by the swarm size, the coordination strategies mentioned above should remain effective after an increase or decrease of the swarm size [108]. In the ant example used above, scalability would be observed as the ability of a few ants to use a similar solution as many ants would. Foraging strategies vary, based on the size of a colony, but the group hunting strategy, for example, has been observed in colonies ranging from 10 000 to 30 000 000 ants. The foraging strategy is therefore scalable to a lesser degree than the group hunting strategy.

These characteristics of robot swarms present high-level goals that natural swarms set for artificial swarms to achieve. Emulating the behaviour observed in insects already presents a substantial challenge, but a much more ambitious project is not observing and emulating ant or termite colonies, but rather humans at the peak level of team sports.

2.5.4 RoboCup

An international scientific initiative in the swarm robotics field is RoboCup [111]. The organisation was formed in 1997 with the mission of developing a team of robots capable of beating the champions of a soccer World Cup by 2050. Figure 2.19 on the following page shows two autonomous robotic players engaged in a soccer match.

2.5.4.1 Robot networking

Vandenbergh et al. [113] presents the possibility of adapting vehicular ad hoc networks (VANETS) to robotic ad hoc networks (RANETs). VANETs are oriented towards low-throughput messaging, whereas RANETS would need to support high throughput streaming media as well. As robots move around in three-dimensional space, a RANET would need to support three topologies. Three examples of robot team use cases are presented in Figure 2.20 on page 51 with their associated topology [113]. The chain network topology is appropriate in cases where individual robots are controlled by human operators. The mesh network topology is appropriate in cases where there are no operators and robots perform their task autonomously. The clustered topology is used when a single human operator is responsible for a group of semi-autonomous robots.

Vandenbergh et al. [113] present an evolution of the wireless networking technology used in robot teams. Initial solutions used infrared communication which allowed for analogue point-to-point



Figure 2.19 Fully autonomous soccer match at RoboCup 2017 [112].

link communication, but infrared is blocked by obstacles and influenced by other light sources or weather. Radio-frequency communication improved upon this by allowing analogue point-to-point links that could penetrate obstacles, but had a low range. The final improvements came with the use of IEEE 802.11 Wi-Fi and IEEE 802.15 Bluetooth which allowed for digital multi-hop links over longer ranges. The network to be used for the artefact presented in Chapter 3 is IEEE 802.11 networking in its mesh configuration (refer to Section 2.2.2).

2.5.4.2 Robot processing

From the inception of the RoboCup challenge in 1997, computation has been part of the challenge. Hedberg [114] noted some of the high-level requirements of building a world-class robotic soccer team:

- capable single players (autonomous agents, robotics, vision and real-time sensor fusion);
- teamwork (multi-agent collaboration, context recognition);
- understanding the competition (cognitive modelling);
- the ability to develop and execute plays and strategies in real time (strategy acquisition, real-time reasoning and planning and reactive behaviour); and
- pre- and post-game training (machine learning).

All of these requirements need varying levels of computation to be satisfied. Jawhar et al. [115] discuss two processing paradigms for use in multi-robot teams:

- Cloud-based systems that leverage cloud computing to provide the necessary compute, storage and advanced software services that can be used to support multi-robot systems. A notable challenge with this approach is latency: results processed on a cloud compute provider's server might not be ready in time for it to be used in a robot team; and

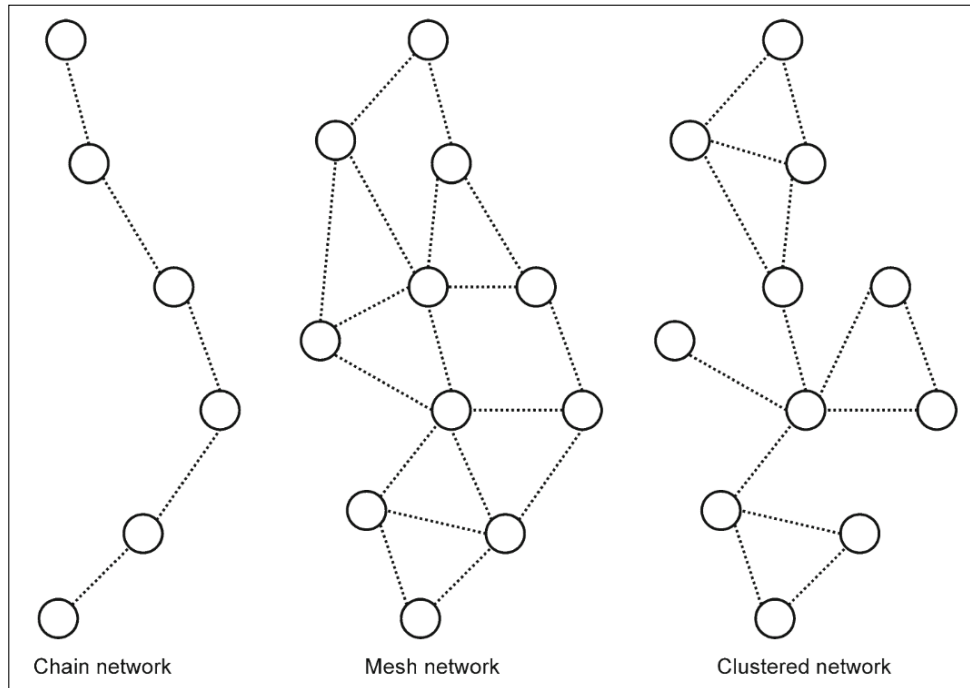


Figure 2.20 Network topologies that have to be supported by a RANET protocol [113].

- Fog-based systems that distribute the computing load between local computers and cloud resources. The most urgent computational work can be performed locally while less urgent and more computationally intensive tasks are processed in the cloud. The use of fog computing is complementary to cloud computing in multi-robot system applications.

Luo et al. [116] focused on object detection using distributed computing that ingested data from the *Microsoft Kinect* camera and notes that ingesting the large amount of data produced by the sensor was impractical on the robot's on-board industrial computer. Their approach used an embedded *Jetson TX1*¹⁶ supercomputer module that processed the information from the Kinect camera. The compute module still was not located on the robot itself.

A goal of this research, however, is to perform all processing on nodes that could feasibly be located on the robots themselves. The reason for this is that by depending on an external computational resource, such as a cloud provider or nearby computer, a single point of failure is introduced. This research also has no requirements to support computer vision or other highly demanding applications. The level of computation power available will therefore be much lower with a focus on enhancing the fault tolerance of the wireless distributed computing system.

2.6 Summary

In this chapter, background information was presented to inform the problem domain analysis and solution analysis processes. The concept of robustness was presented first and increasing fault

¹⁶<https://developer.nvidia.com/embedded/jetson-tx1>

tolerance in particular was presented as a suitable method of improving it. System faults were introduced and classified along with fault-detection mechanisms and approaches to improve fault tolerance.

The three central theoretical concepts were presented: wireless networking, distributed computing and system architecture. As the title of the research suggests, the platform developed needs to be suitable for a robot swarm application and the chapter was concluded with an investigation into the desirable characteristics of robot swarms, as well as a view of robot networking and processing from the RoboCup research initiative.

In terms of the engineering cycle presented in Section 1.6, the literature review supports knowledge questions, whereas the practical efforts undertaken in the next chapter solve design problems. The three central concepts support the research solution developed in the next chapter as presented in Figure 1.3 on page 9. The theoretical insights produced by reviewing existing literature will be extended in the next chapter by practically applying systems architecture to create a wireless distributed computing platform.

Chapter 3

Artefact Creation

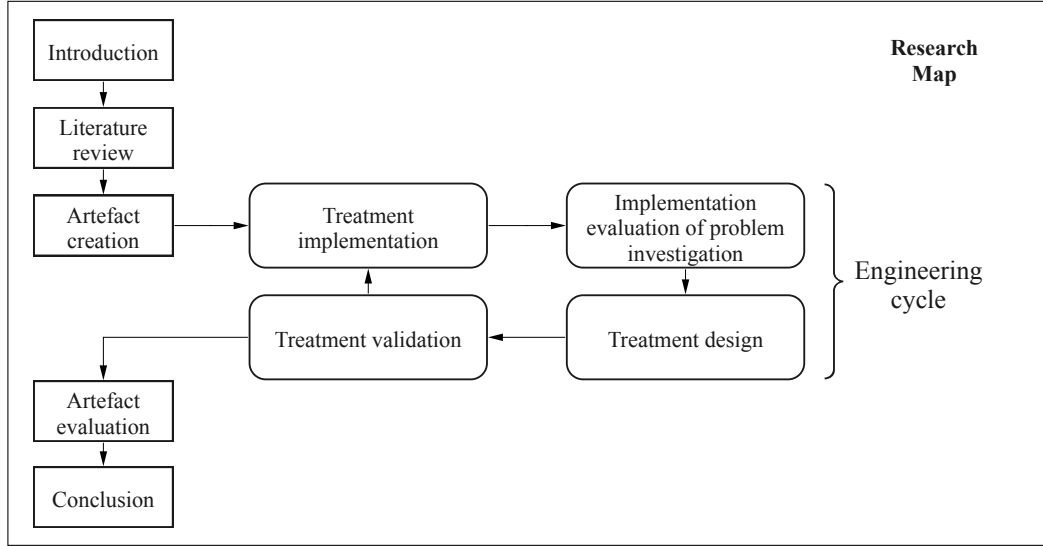


Figure 3.1 Research map extract for Chapter 3. The complete research map is presented on page 1.

In the previous chapter, existing literature was reviewed to provide a theoretical foundation for the practical efforts of this research. In reference to the steps and guidelines presented in Section 1.6.1, this chapter represents step 3 in the DSR approach: design and development. The development of the artefact is driven by a systems architecture approach in which iterative prototypes of key components in the system are developed and evaluated, using an illustrative scenario (refer to Section 1.6.3). In this chapter, the illustrative scenario application that will be used to evaluate fault tolerance will be presented. The artefact’s design consists of hardware, networking and software components and are presented in Sections 3.2, 3.3 and 3.4. The arrangement of these components represent the DSR method (see Section 1.6.2) of supporting fault-tolerant wireless distributed computing.

3.1 Illustrative scenario

Iterations of the artefact are compared in respect of how they contribute to fault tolerance in a chosen scenario: a simulated drone swarm. This choice of scenario is driven by the research title: “*A fault-tolerant wireless distributed computing platform for robot swarms*”. The Intel demonstration, presented as the inspiration for this research in Chapter 1, used a very sophisticated flight control system in order to perform its function. However, all processing was performed at a central point as shown in Figure 1.2 on page 4.

By simplifying the requirements of the flight control system, the challenge of performing this functionality using wireless distributed computing can be undertaken. The concept is presented in

Requirement	Description
Cost	The hardware platform needs to be low-cost so multiple units can be purchased.
Power	Units must consume a low amount of power so they can be battery-powered.
Processing	Units need sufficient computational power to validate distributed computing.
OS	Units need to run Linux, as it is free and widely used in technical artefacts.
Networking	Units need to support IEEE 802.11s wireless mesh networking.

Table 3.1 Requirements set for selecting the hardware platform on which the artefact will be built.

Figure 2.13 on page 38 and represents a robot team, introduced in Section 2.5. Each drone includes sensing, compute and networking capabilities and can work towards a common goal. Creating an actual drone swarm is outside the scope of this research, but a simplistic flight control system can be developed with two aims:

1. prevent drones from “colliding” by enforcing a minimum distance threshold; and
2. prevent drones from moving out of transmission range by enforcing a maximum distance threshold.

The focus is to establish the validity of performing processing in a distributed fashion on the drones themselves without relying on any centralised or remote computational resources. The base hardware for the eventual platform was determined first, presented in Section 3.2. The mesh networking solution is selected in Section 3.3. On top of the hardware layer and networking solution, several iterations of distributed application software are built and are presented in Sections 3.4.1 through 3.4.4. The fault tolerance of each iteration is evaluated using the described illustrative scenario. The result of this iterative approach is the final artefact of this research, presented in Section 3.6, which will be the subject of further evaluation in Chapter 4. The artefact’s design is the end result of the system architecture process undertaken in this chapter.

3.2 Platform hardware

The first step in the development of the artefact was to choose a hardware platform. Conceptually, the hardware layer represents the base of the platform on which to build. Design choices are made as part of the systems architecture process and decisions made at this level have the greatest impact on the options available for decisions at higher levels. This choice was guided using a set of requirements for the hardware platform, presented in Table 3.1.

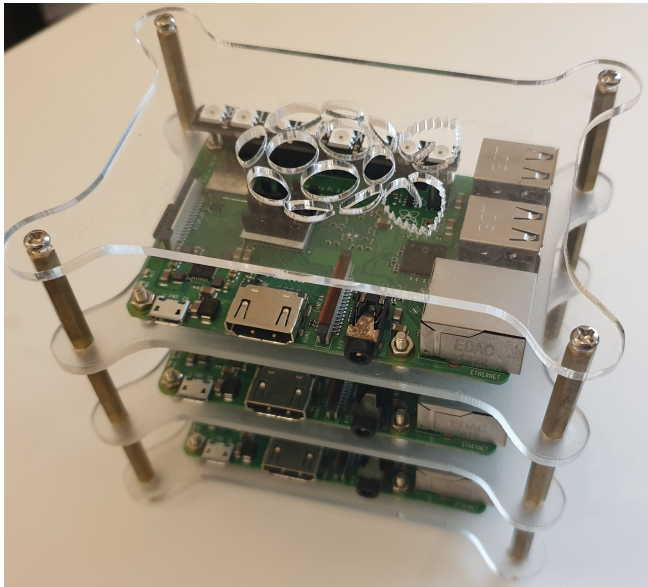
From this set of requirements, an appropriate choice was the Raspberry Pi¹ single-board computer (SBC). There are many alternative SBC available including the ASUS Tinker Board² and ODROID XU4³. The Raspberry Pi was chosen due to its price, availability and familiarity. The latest Raspberry Pi model available when the design choice was made was the Model 3 B+, while the

¹<https://www.raspberrypi.org>

²<https://www.asus.com/us/Single-Board-Computer/Tinker-Board>

³<https://www.hardkernel.com>

Model 4 B has been released since. It is an affordable option, which made it possible to purchase multiple units, as shown in Figure 3.2a. The Model 3 B+ consumes ~ 350 mA (1.9 W) at idle and ~ 980 mA (5.1 W) at full load [117]. This low power usage enables the computer to be run from a mobile power bank, which makes the computer mobile and a feasible payload on a drone. The Raspberry Pi is officially supported by multiple Linux distributions, including Debian, Ubuntu and Arch Linux. In terms of memory and computational resources, the SBC is equipped with 1024 MB of RAM and a quad-core BCM2837 System on Chip (SoC) running at a maximum frequency of 1.4 GHz. Finally, the SBC has on-board wireless networking hardware supporting the IEEE 802.11 standard in both the 2.4 GHz and 5 GHz bands. Unfortunately, the on-board networking hardware does not support the required mesh-point mode needed to construct a mesh network. Fortunately, mesh-point functionality was enabled using an external USB network adapter, as shown in Figure 3.2b.



(a) Platform compute: three Raspberry Pi Model 3B+ units in an acrylic “rack cluster”



(b) Platform networking: mesh-point capable USB network adapter with the replacement cable

Figure 3.2 The combination of computation and networking hardware used to construct the platform. Note that for the experiments presented in Chapter 4, the units were separated from this rack to ensure a specific distance between network adapters. This was necessary due to the short length of the replacement cables.

3.3 Platform networking

Based on the literature investigation on IEEE 802.11 wireless networking modes (presented in Table 2.6 on page 23), the IEEE 802.11s “*mesh networking*” standard was chosen and incorporated in the design of the platform. Alternative mesh networking solutions that were considered include

ZigBee⁴ (IEEE 802.15.4) and LoRa⁵ (Long Range). The main rationale for the choice to use IEEE 802.11s is that both alternatives have significantly lower data-rates: 120 kb/s and 50 kb/s, respectively [118], [119]. As the platform needs to facilitate distributed computing, higher data rates will be required.

In the classification presented by Kurose and Ross [18], the chosen implementation has to support multi-hop transmissions and the network does not contain any base stations. The motivation behind this choice is to eliminate single points of failure and produce a decentralised architecture that improves fault tolerance [42]. If the network used IEEE 802.11n or IEEE 802.11ac in infrastructure mode, the entire system would depend on an access point being accessible. If this access point should fail, the platform would also fail, regardless of mitigations made at higher levels. The platform would therefore not be fault tolerant. The IEEE 802.11s standard does not specify a routing protocol, but several are available, including [120]:

- Open80211s;
- BATMAN (better approach to mobile ad hoc networking);
- BATMAN-adv (BATMAN Advanced); and
- OLSR (optimised link state routing).

Pojda et al. [120] compared the goodput of these four protocols, as shown in Figure 3.3 on the following page. Goodput is defined as “*the ratio of the expected delivered data payload to the expected transmission time*” [121] and represents the throughput achieved after all overheads have been taken into account. Pojda et al. found both Open80211s and BATMAN-adv to be mature enough to satisfy performance requirements in “*harsh environmental conditions*”.

Attempts at creating a mesh network using the *open80211s* package on the Raspberry Pi hardware were not successful. Work by Jünemann [122] suggests that this could be achieved, but his result could not be replicated on the available combination of hardware and software. Limited analysis of the problem points to the *rt2800usb* kernel module driver. Furthermore, his work was performed in 2013 using a much older version of the Linux kernel (4.4.13) on older hardware. Kernel support for the Raspberry Pi Model 3 B+ is only available from version 4.14, so the possibility of using this older version of kernel was not explored.

Fortunately, creating a mesh network using the *batman-adv* package on the Raspberry Pi platform was successful. The *batman-adv* kernel module is included in the stable Linux kernel which is maintained by the Raspberry Pi foundation⁶. Based on this support and the insignificant loss in performance compared to *open80211s* (see Figure 3.3 on the following page), the choice was made to create a mesh network using BATMAN Advanced. A script was developed to configure the mesh network on start-up, presented in Section B.1.3.

⁴<https://zigbeealliance.org/>

⁵<https://lora-alliance.org/>

⁶<https://github.com/raspberrypi/linux>

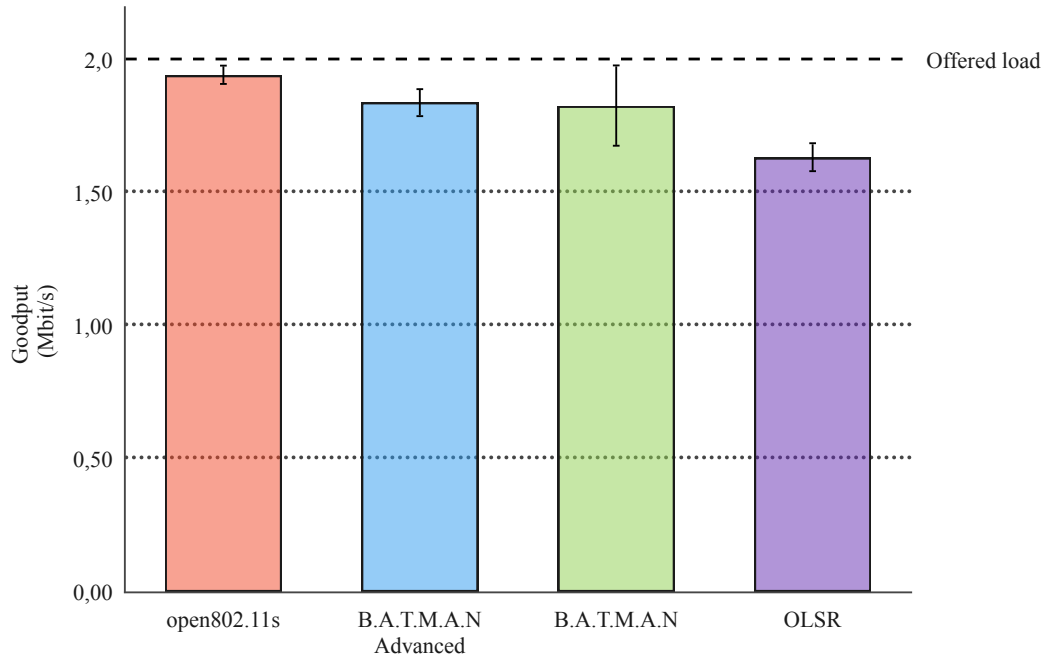


Figure 3.3 Comparison between mesh routing protocols. Figure redesigned from the work of Pojda et al. [120].

3.4 Platform software iterations

In this section, the design iterations of the artefact leading up to the final design of the artefact are presented in a sequential format. In practice, however, several avenues of investigation were taken and often in parallel. Challenges faced while pursuing one implementation informed the choice of which implementation to attempt next and solutions discovered later in the process enabled further investigation into earlier iterations. Every design iteration of the artefact, including attempted solutions that are not presented in this chapter, can be found in the public git repository⁷ for the research.

In this section, an overview of the iterative development of the artefact is presented. Each iteration is presented by providing an overview of the core technologies it uses, how it is implemented and an evaluation of its fault tolerance. Iterations 2 through 4, presented in Sections 3.4.2, 3.4.3 and 3.4.4 respectively, all present improvements in fault tolerance beyond what was delivered by their predecessor. The first iteration does not have fault tolerance as a goal. Instead, it is focused on establishing the feasibility of a wireless distributed computing platform with the given hardware, software and networking.

⁷<https://github.com/prossouw79/research-artefact>

3.4.1 Iteration 1: MPICH

The message-passing-interface (MPI) standard is a well-established approach for creating distributed computing applications [123]. In the first iteration, the aim was to establish a baseline for distributed computing on wireless networks. The chosen approach is to create a high performance computing cluster (HPCC), introduced in Section 2.3.2.1 on page 35 with the application architecture presented in Figure 3.4.

In this initial iteration of the platform, two goals were set:

1. Establish compatibility between an MPI environment and the *armv7l* processor architecture of the Raspberry Pi; and
2. Establish compatibility between an MPI environment and the wireless mesh network created using *batman-adv*.

The combination of wireless mesh networking with MPI distributed computing seems to be an under-explored area in literature and therefore it was attempted in this iteration. Both of the goals were achieved; it is possible to create an MPI distributed computing environment with Raspberry Pi machines communicating over a wireless mesh network.

Since MPI is so well-established within distributed computing, mature tools are available that function within an MPI environment. The OSU micro-benchmark suite⁸ is developed and maintained by the network-based computing laboratory at Ohio State University, alongside the MVAPICH MPI library/runtime [124]. The benchmark suite is compatible with multiple MPI implementations, including the MPICH environment used in this iteration of the platform.

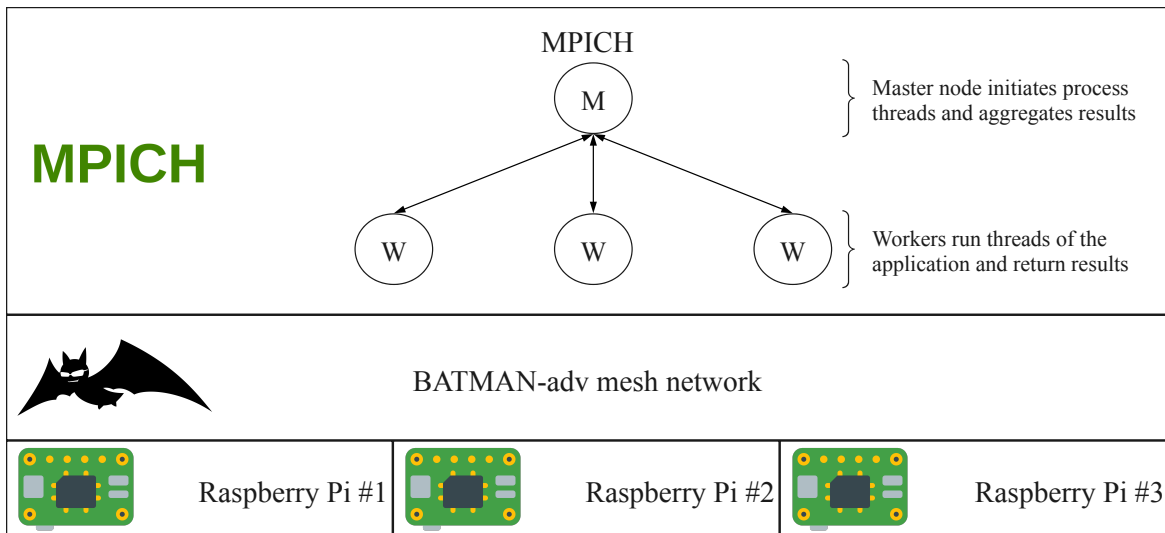


Figure 3.4 High-level architecture for the MPICH implementation.

The OSU micro-benchmark suite provides many MPI communication benchmarks with two classes

⁸<http://mvapich.cse.ohio-state.edu/benchmarks/>

of benchmarks of particular interest to this research: point-to-point and collective tests to stress different parts of a cluster system. By running point-to-point tests in two threads on separate nodes over the mesh network, the effective performance of the network can be measured. Collective tests measure the QoS metrics for traffic between multiple threads distributed over all participating nodes. The metrics measured by this class of benchmarks are more reflective of the distributed system as a whole, and therefore, metrics are presented as averages.

Unfortunately, while the combination of MPI and the mesh network is functional, its lack of robustness in challenging environments became evident before testing with OSU could be performed. The external USB network adapters used to create the mesh network used very low-quality USB cables with connectors that were prone to losing connection with the slightest movement. When this happens, the *batman-adv* kernel module does not automatically reconfigure the adapter to rejoin the mesh network and no MPI application, e.g. the OSU micro-benchmark suite, will continue. In reference to Table 2.1 on page 16, this implementation of MPI does not demonstrate fault tolerance or recoverability. The cable-disconnection issue was rectified by replacing all the cables for the adapters with higher quality cables. However, the development of this iteration revealed two important concerns:

1. an MPI distributed computing environment is not self-healing or auto-scaling; if a node is briefly disconnected or a new node is connected, MPI does not respond accordingly; and
2. while *batctl* is used for configuring the mesh at first, it also does not detect or manage reconnecting adapters.

Addressing the first concern would require an effort beyond the scope of this research, but the second concern can be alleviated with scripting logic. A shell script can run continuously on a node and routinely count its neighbour nodes. If the number of neighbours is zero, it will assume that it is the one that has been disconnected and reinitialise its mesh connection. In practice, the replacement cables proved sufficient for ensuring that the network adapters are not disconnected. Insights gained from constructing and analysing this iteration of the platform include:

- The MPICH implementation of the MPI standard can function over a mesh network and is compatible with the *armv7l* processor architecture of the Raspberry Pi SBC;
- The OSU micro-benchmark suite is well suited to measuring QoS metrics in a distributed computing environment;
- The *batctl* CLI tool is not capable of detecting network interfaces that are temporarily disconnected. This compromises the recoverability aspect of platform robustness presented in Table 2.1 on page 16; and
- MPI does not scale dynamically during execution. Scaling the cluster up or down requires reconfiguration. This trait is a significant barrier to the scalability aspect of platform robustness presented in Table 2.1 on page 16.

These insights informed the design choices made for the next iteration of the platform. It is clear

that the MPI standard, while well-established in the distributed computing field, is ill-suited to this dynamic environment. An attempt was made to increase the fault tolerance of the MPI environment using containerisation (refer to Section 2.4.4). The MPI environment is heavily dependent on its configuration and does not fail gracefully when the configuration does not match the actual environment (see Figure 2.2 on page 17). The work of Nguyen and Bein [125] used the Docker Swarm container orchestrator to dynamically configure and run an MPICH cluster as a collection of scalable container services. The paper won the Best Paper Award at IEEE CCWC 2017 and the result of their work is available on their GitHub repository for *alpine-mpich*⁹. The *alpine-mpich* project was forked¹⁰ for this research and adaptations were made, including:

1. The project supports both *x86_64* and *armv7l* processor architectures. This allows for the creation of scalable clusters on the Raspberry Pi as well as general *x86* machines;
2. The node-detection mechanism was changed to no longer rely on a periodically updated machines file. Instead, the node-detection functionality was implemented at the application level using an automated network scan before a job was run;
3. the master-node service was changed to be a global service. One master-node was therefore available on every swarm node and this ensured that in the event of a node being disconnected during operation, only the worker-node containers that were not running on the same host would be unreachable, and a processing job could be restarted on the nodes that were left. It could not, however, recover the interrupted job;
4. The OSU micro-benchmark suite was added to the image for performance analysis and an automated benchmark running script was developed that ran a configurable set of OSU benchmarks. The script also collects, parses and plots the results of every benchmark run and stores the results on AWS S3¹¹ storage for easy access.

The result of these modifications is a functional, dynamically scalable cluster running as containers on the mesh-connected Raspberry Pi platform. The cluster can be scaled up or down using simple commands developed in the original work. While containerisation improves the robustness of the MPI framework through simplified replication, it could not solve the primary issue preventing robustness: the MPICH cluster is not self-healing or fault tolerant. Although the cluster can scale dynamically and the cluster configuration was automatically updated, it cannot do so while a processing job is running. When scaling up, the running job is not interrupted, but it does not utilise the newly available resources either. Scaling the cluster down interrupts the running job as processing threads were running in the containers that were terminated. Disconnecting any node during processing immediately halts the active processing job. For these reasons, this platform cannot be considered robust in dynamic environments, as it does not demonstrate sufficient fault tolerance. This method was not developed further.

⁹<https://github.com/NLKNGuyen/alpine-mpich>

¹⁰<https://github.com/prossouw79/alpine-mpich>

¹¹<https://aws.amazon.com/s3>

3.4.2 Iteration 2: WebSockets

In this iteration, it is important to improve on the lack of fault tolerance provided by the MPI standard for this application. Based on the guidelines for DSR research, presented in Section 1.6, it is also important to make this iteration relevant to a specific business problem. As an exercise in creating both the platform and a relevant application intended to run on it, this iteration of the artefact includes the simple drone swarm simulation application presented in Section 3.1. The software used to facilitate distributed computing was chosen based on the following requirements:

1. The software needs to function with minimal configuration;
2. The software needs to automatically recover from lost network connections; and
3. The software needs to facilitate distributed computing.

Based on these requirements, two software technologies were selected to investigate: WebSockets and Remote Procedure Calls (RPC). This section presents the iteration based on WebSockets, while the RPC iteration is presented in Section 3.4.3.

The iteration implementing WebSockets attempts to solve problems encountered in the MPI implementation by continuously replicating the application state to participating nodes using an application-level multi-casting approach [42]. Socket connections provide real-time, duplex connectivity between machines and the WebSocket protocol brings this functionality to the web. Two popular open-source libraries associated with WebSockets include Socket.io¹² and Microsoft's .NET Core SignalR¹³. Leveraging the duplex communication provided by WebSockets makes it simple to keep the application state synchronised across all participating nodes in several situations, summarised in Table 3.2.

Situation	Description
On application start	When the first node starts the application, it constructs the first state and starts processing.
When a new node is added	For every consecutive node that is added, the first node can broadcast the entire application state to it.
During execution	When nodes are cooperatively accomplishing their task and modifying the application state, these changes are communicated to all other nodes.
When a node disconnects	If a node disconnects, every other node will be able to detect that it is not reachable with a simple health check (see Section 2.1.2).
When the node reconnects	Once the disconnected node becomes available again, its local version of the state can be compared to the distributed state and changes merged.

Table 3.2 State management in various situations using WebSockets.

¹²<https://github.com/socketio/socket.io>

¹³<https://github.com/SignalR/SignalR>

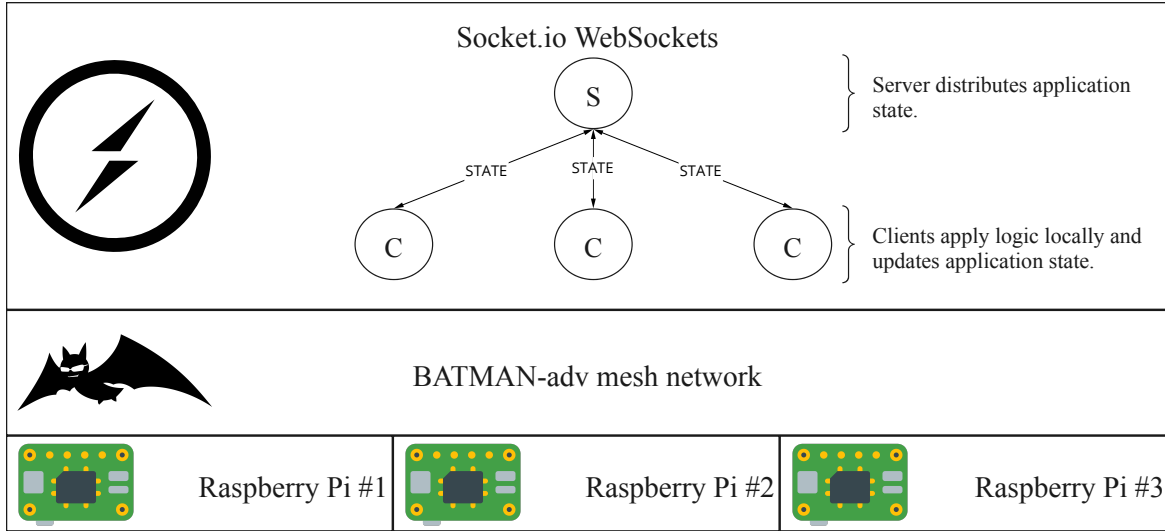


Figure 3.5 High-level architecture for the WebSocket implementation.

This approach as presented in Figure 3.5 is fundamentally different from the MPI approach (see Figure 3.4 on page 58), where processing threads synchronise intermediate results of execution. In the MPI paradigm, no single node is required to have a complete version of the distributed state. Not even a master node running the application has a complete representation of the distributed state. This allows an MPI environment to process massive data-sets that cannot be stored in any single node’s memory resources, at the cost of fault tolerance in the event of a node failure.

The WebSocket approach sacrifices the ability to handle such large jobs for the ability to recover from a node failure with replication. This approach improves fault-tolerance by maintaining synchronised replicas of the application state. An application dashboard was also developed to visualise the drone swarm’s movement over time. Extracts from the application dashboard are presented in Figures 3.7 to 3.8 on page 64. In Figure 3.7, the relative positions and location history (indicated by trails) are shown in a three-dimensional plot, while Figure 3.8 shows the measured distances between drone pairs and the average speed.

This application was developed using the socket.io implementation of the WebSocket protocol. The WebSocket server and clients were developed in JavaScript, targeting the Node.js application runtime. The server created a socket host to which all clients (simulated drone nodes and application dashboard) connected. The application was deployed on the Raspberry Pi nodes, connected with the mesh network and a single machine also hosted the dashboard, which was developed using AngularJS¹⁴, plotly.js¹⁵, and Bootstrap¹⁶.

While the application is distributed, based on Tanenbaum’s [42] definitions of distributed systems presented in Section 2.3.1, this iteration closely resembles a distributed information system, rather

¹⁴<https://angularjs.org/>

¹⁵<https://plot.ly/javascript>

¹⁶<https://getbootstrap.com>

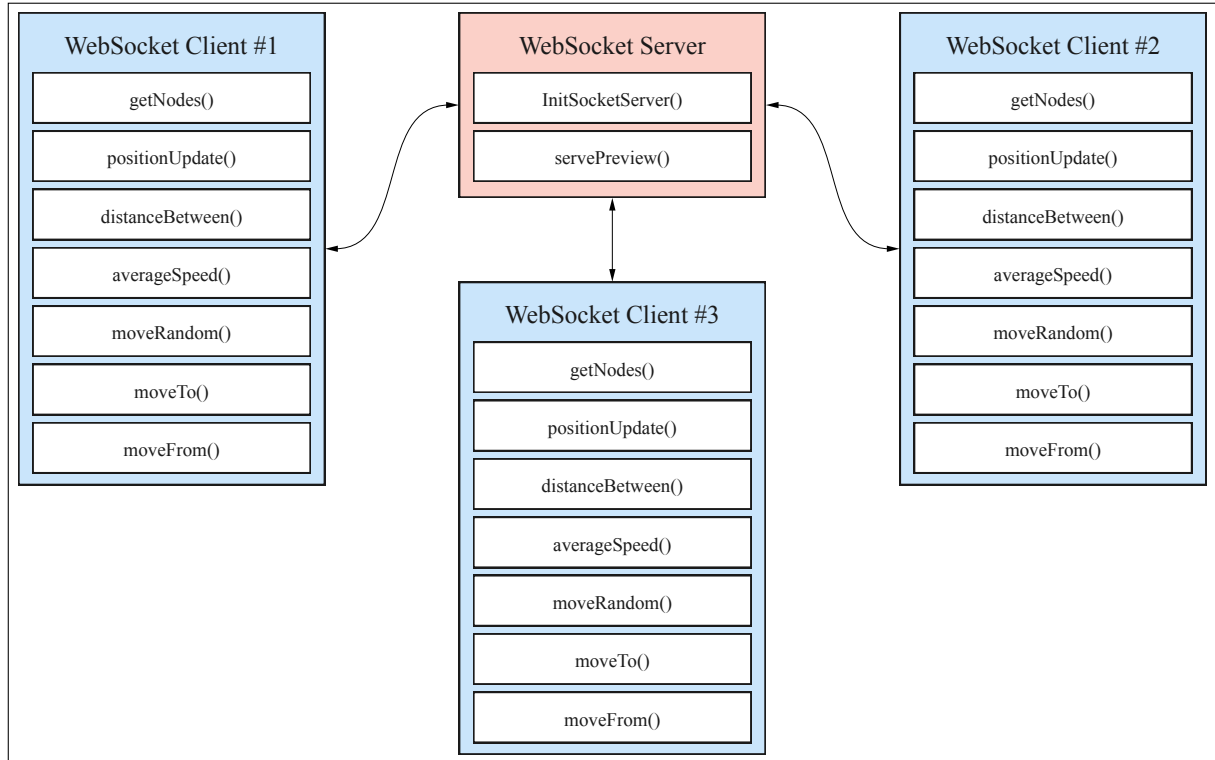


Figure 3.6 Software components of the WebSocket drone swarm application.

than the desired distributed computing application. Figure 3.6 shows a representation of the software components. Each node stores a local copy of the application state, modifies it to include the changes to the environment and synchronises it back to the WebSocket server to update every other node. Each client still needs to run all application logic on a local thread and is only focused on determining the changes to the distributed state that concerned its own movement. For this reason, it only satisfies the first two of the stated goals for this iteration of the artefact: it requires minimal configuration and can recover in the event of any node temporarily disconnecting.

While this implementation has looser coupling between nodes than the MPI implementation, it still depends on the single WebSocket server node. If this host becomes unavailable, all communication is halted until a connection can be re-established to this node. No alternative node can take over the responsibility of hosting the WebSocket server in the event of a failure, i.e. there is no redundancy to facilitate fault tolerance. While the iteration as a whole presents improvements over the MPI implementation, it still includes a single point of failure and prevents this implementation from being sufficiently fault tolerant. The next iteration will build on the insights gained to improve fault tolerance.

3.4.3 Iteration 3: RPC

The remote procedure call (RPC) implementation applied a different protocol and paradigm to the drone swarm application created in Section 3.4.2. In the WebSocket implementation, each node

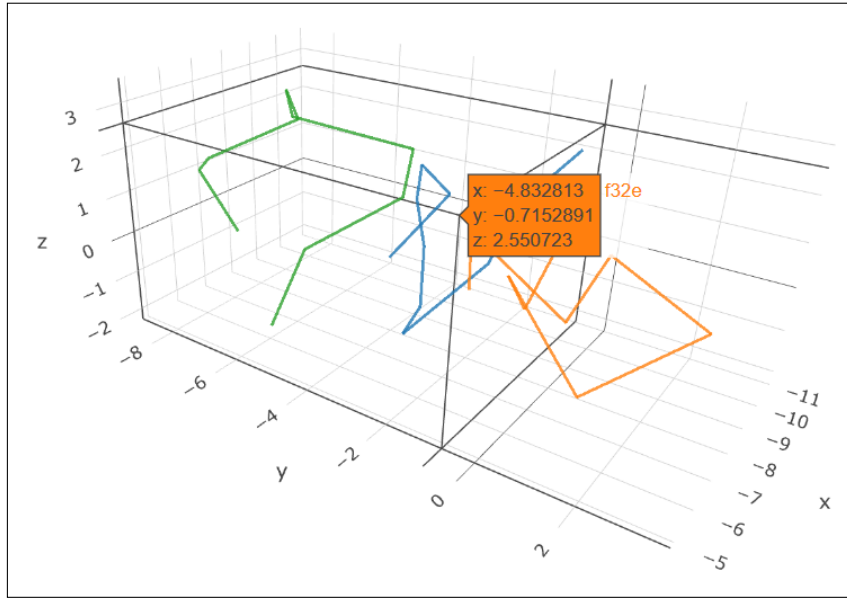


Figure 3.7 Extract from drone swarm application dashboard: 3D position plot with trails.

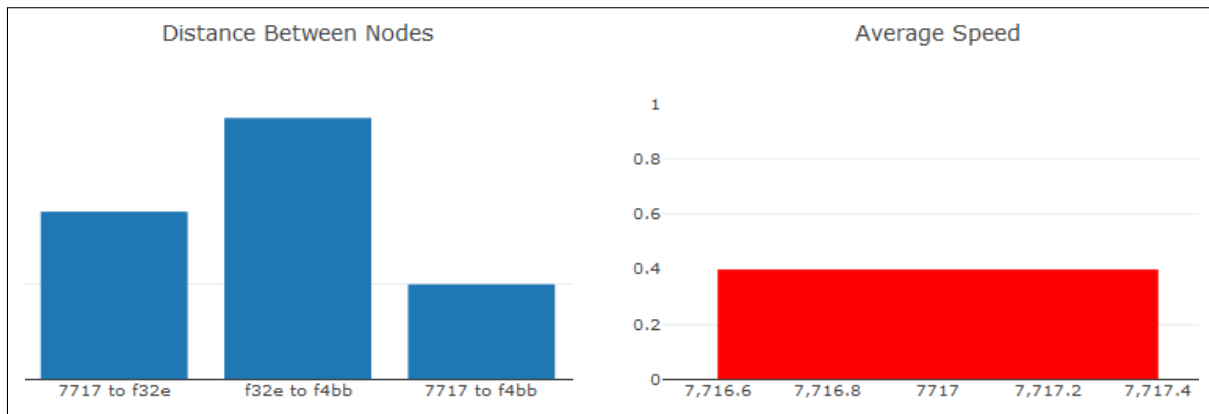


Figure 3.8 Extract from drone swarm application dashboard: distance between nodes and average speed.

applies local logic to distributed data. The remote procedure call (RPC) implementation aims to decouple logic and data and facilitate distributed computing.

The JSON-RPC protocol is a lightweight, transport-agnostic and stateless RPC protocol [126]. Being transport-agnostic refers to the characteristic that the protocol's implementation is not tied to a specific transport protocol; it can be implemented within the same process, over WebSockets, over HTTP, or in various message-passing environments. This version of the application was built using the jayson¹⁷ implementation of the JSON-RPC 2.0 protocol¹⁸ and used HTTP to provide transport functionality.

A design decision was made to decouple the application into two separate applications: compute

¹⁷<https://github.com/tedeh/jayson>

¹⁸<https://www.jsonrpc.org/specification>

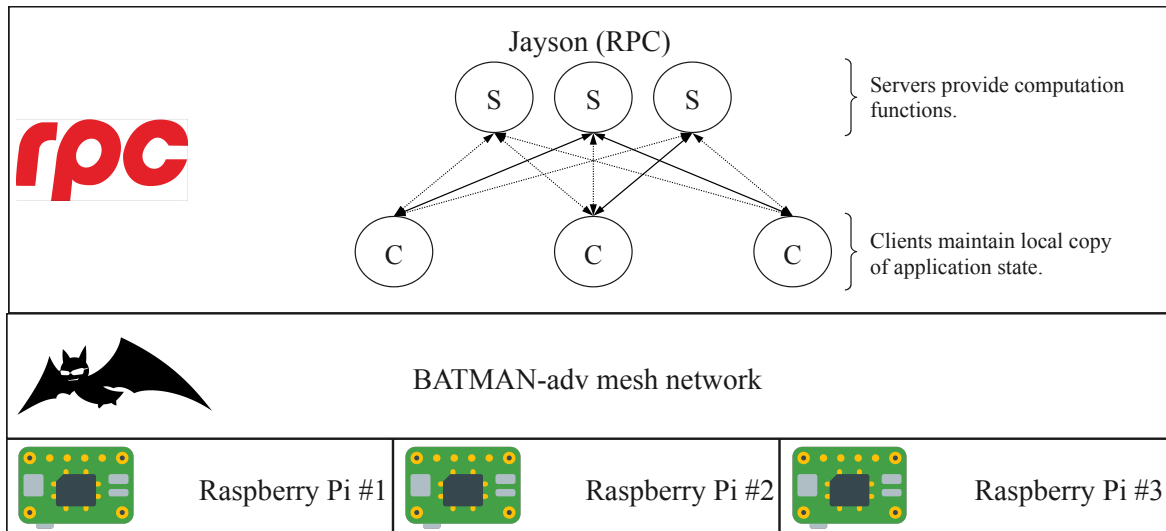


Figure 3.9 High-level architecture for the Jayson RPC implementation. Compute servers are marked *S* while clients are marked *C*. The arrows represent function calls, with each client being able to call functions from any server with load balancing.

servers and clients as presented in Figure 3.9. The server application exposes functions that the clients could call over the network. An instance of the client application represents a drone in the same way as the previous iteration - a unique identifier and position in three-dimensional space. Each client still stores a local copy of the application state, but it no longer needs to perform transformations on it locally.

Once a server instance is started on one or more of the available nodes, it first initialises an HTTP transport service that exposes its functions. It then measures and reports the number of requests it responds to every second. When a client instance starts up, it establishes an HTTP connection with every available server instance. Once connected, it will request its globally unique identifier from a compute server, which from this point onward identifies this client among all other clients. The client will then start periodically sending its position (state) to a randomly selected compute server and requesting the distance between it and other nodes. The server computes this information and returns it to the client. Based on these positions, the client will request either to get coordinates that will move it further from or closer to a node. If all other nodes are within the set distance thresholds, it requests a random movement from its current for it to move towards. The architectural representation of this application is presented in Figure 3.10 on the following page.

Each of the compute client instances is aware of all available compute servers, and sends its requests to the servers in a simple load-balancing scheme. The design of the application does not guarantee that each request is received or responded to, but avoids a catastrophic failure of the application when individual requests are not responded to. The fault tolerance of the application is enhanced by implementing asynchronous execution through JavaScript Promises.

A Promise “represents the eventual completion (or failure) of an asynchronous operation and its

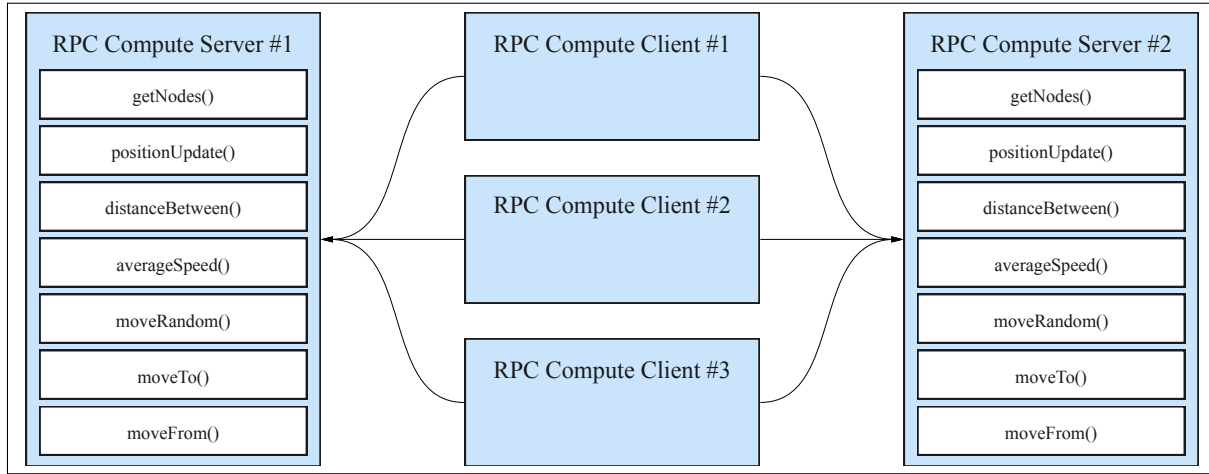


Figure 3.10 Architecture for the RPC implementation. Multiple decoupled compute servers provide service to independent clients. Two compute servers are presented in this figure, but the system can run with one, two or more compute servers.

resulting value” [127] and is a response to the need of modern web applications to perform work asynchronously. Every function call to the RPC server returns a Promise object to be handled if and when the server responds with the result data. While the client is waiting for this response to resolve, it is free to do other processing. Figure 3.11 on the following page presents the three states of a Promise. The pending state indicates that the request has been instantiated, not necessarily that it has been received by the server. The Promise state changes to fulfilled once the server has responded with the result of the request, or rejected in the case of an error.

It is important to note that Promises can remain in the pending state indefinitely. It is possible that the server never resolves or rejects the promise, but since the client is not blocked by waiting for the result, it does not stop the client from functioning. The asynchronous processing concept was implemented in this iteration using JavaScript Promises and proved to be key for creating fault-tolerant distributed systems: wherever possible, a node should be able to continue functioning while waiting for a response.

Compute clients in this implementation use non-blocking routines so that in the event of a node or transport failure, a thread is never blocked. If a compute node fails or is disconnected, it only affects that individual request - not the application as a whole. From an architectural perspective, this approach is very different from the MPI paradigm in which functions like *MPI_Barrier* are used specifically to block remote threads until all local and remote processing threads reach the current routine. By allowing nodes to function independently from one another, asynchronous execution greatly enhances the fault tolerance of a distributed system.

This design of the application improves upon the WebSocket implementation by decoupling state and processing and can be characterised as a distributed computing system and not only a distributed information system. Furthermore, the significance of asynchronous execution to the fault tolerance

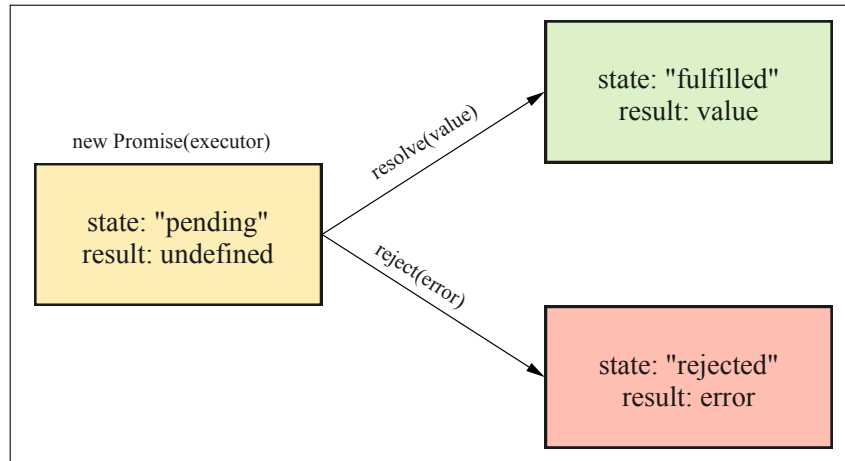


Figure 3.11 Asynchronous execution: states of a JavaScript Promise [127].

of distributed systems was encountered in practice while developing this application.

The WebSocket application also depends on a single host to initialise and maintain the connection between clients, whereas the RPC application performed the same function without this single point of failure. Without this requirement for a single host, an instance of the compute server can be run on each participating node. Each of these instances can serve requests from nodes in the network, including the client instances that might be running on the same host. This approach improves fault-tolerance by making distributed computation highly-available through replication.

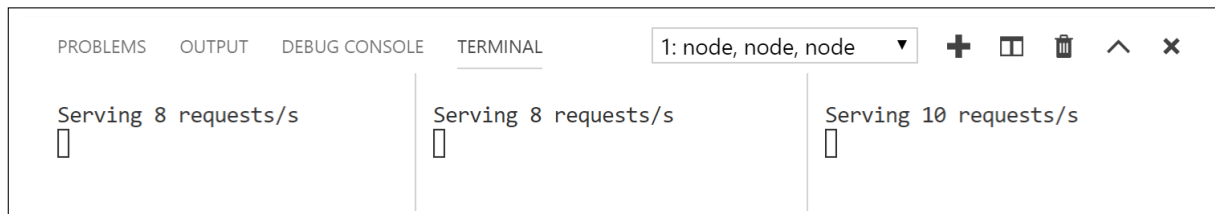


Figure 3.12 Three instances of the RPC jayson compute server, reporting the number of requests served per second.

Figure 3.12 shows three instances of the jayson compute servers running on a local machine. Each instance exposes asynchronous functions that can be called over the network. The servers simply output the number of requests they are responding to every second. Figures 3.13 and 3.14 present two potential states for compute clients. The first state is during runtime when all compute servers are available. Each client calls the functions exposed by the compute servers and prints a code corresponding to the called function e.g. *BC_POS* for *broadcastPosition()*. In Figure 3.13 on the following page, a corrective action is highlighted on each node in red, green and blue respectively. The section indicated in red shows that the node with GUID *7a3a-78fd-d7c6* is moving closer to node *fa62-aa75-9b54*. *MV_RND* indicates that the node is not taking corrective action; it is moving randomly.

7a3a-78fd-d7c6 < -- < fa62-aa75-9b54 BC_POS MV_RND GT_DST GT_DST 7a3a-78fd-d7c6 > -- > fa62-aa75-9b54 BC_POS MV_RND GT_DST GT_DST 7a3a-78fd-d7c6 > -- > fa62-aa75-9b54 BC_POS MV_RND GT_DST GT_DST 7a3a-78fd-d7c6 > -- > fa62-aa75-9b54	f252-c732-f029 < -- < 7a3a-78fd-d7c6 BC_POS MV_RND GT_DST GT_DST f252-c732-f029 < -- < 7a3a-78fd-d7c6 BC_POS MV_RND GT_DST GT_DST f252-c732-f029 < -- < 7a3a-78fd-d7c6 BC_POS MV_RND GT_DST GT_DST f252-c732-f029 < -- < 7a3a-78fd-d7c6	GT_DST BC_POS MV_RND GT_DST GT_DST fa62-aa75-9b54 > -- > f252-c732-f029 BC_POS MV_RND GT_DST GT_DST fa62-aa75-9b54 > -- > f252-c732-f029 BC_POS MV_RND GT_DST GT_DST fa62-aa75-9b54 > -- > f252-c732-f029
7a3a-78fd-d7c6 > -- > fa62-aa75-9b54	f252-c732-f029 < -- < 7a3a-78fd-d7c6	fa62-aa75-9b54 > -- > f252-c732-f029

Figure 3.13 Three instances of the compute client instance, each printing which method it is requesting and any correctional action it is taking. Nodes are identified with their unique IDs and each of the indicated areas shows an example of a corrective action being taken.

GT_DST (node:12512) UnhandledPromiseRejectionWarning: Error: connect ECONNREFUSED 127.0.0.1:3001 at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1106:14) (node:12512) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). (rejection id: 124) GT_DST	(node:8944) UnhandledPromiseRejectionWarning: Error: connect ECONNREFUSED 127.0.0.1:3001 at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1106:14) (node:8944) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). (rejection id: 119) MV_RND GT_DST GT_DST	(node:384) UnhandledPromiseRejectionWarning: Error: connect ECONNREFUSED 127.0.0.1:3001 at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1106:14) (node:384) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). (rejection id: 109) GT_DST GT_DST 6452-0896-6366 > -- > fa2c-d1a1-5234
--	--	--

Figure 3.14 Client response when a compute server becomes unavailable. Note that after the error (marked red), the next request is fulfilled (marked blue) by another compute node. The servers are running locally in this example.

While this represents a simplistic flight control mechanism for a drone swarm, the computation for it is performed in a distributed fashion and does not rely on any single node's availability. In Figure 3.14, one compute server instance was deliberately stopped to see how the client applications would respond. When the client attempts to make a connection to the stopped node, it encounters an *UnhandledPromiseRejectionWarning* (highlighted in red). Crucially, this does not stop the execution, as shown in the next output from the clients, e.g. *GT_DST* (highlighted in blue). The client then requests a distance calculation, and it is fulfilled by a different compute server. Once the stopped compute server instance is restarted, the clients can complete the connection once again, and the system continues at full capacity.

This behaviour represents several enhancements over previous iterations:

- unlike MPI, when a node becomes unavailable, the system continues at a lower capacity;
- unlike the WebSocket implementation, the platform no longer depends on any single node; and
- unlike MPI or the WebSocket implementation, when a previously unavailable node becomes available, clients automatically start utilising it again.

These advantages are the consequence of designing for failure. Using asynchronous execution allows

participating nodes to continue performing functions while waiting for remote processing to finish. Decoupling of processing and data allows for greater redundancy using the replication approach to fault tolerance, presented in Section 2.1.3.2. Replication was performed by running three instances of the compute server (one per node) and allowing the clients to request functionality from any one of them. When one of the nodes fails, the application can continue as presented in Figure 3.14 on the previous page

In this iteration, the goals initially set for WebSocket iteration in Section 3.4.2 were achieved. Like the previous WebSocket implementation, the RPC application could recover in the event of temporary node unavailability and functioned with minimal configuration.

Unlike the WebSocket implementation, however, the RPC application can be considered a viable distributed computing platform as application state transformation was performed remotely. Instead of replacing the single WebSocket server with a single RPC Server, the RPC servers were replicated across all nodes to enhance fault tolerance. The application was also designed to balance its requests across the available nodes and was able to continue operation as long as a single compute server was available. This iteration of the artefact revealed the importance of asynchronous execution, service replication and application design.

3.4.4 Iteration 4: Streaming implementations

In this section, the fourth iteration of the artefact is presented. The drone-swarm simulation application developed in the previous iteration of the artefact demonstrated the importance of asynchronous execution in distributed systems with unreliable connections. Fault tolerance was increased by designing the application in such a way that it could respond asynchronously to client requests without blocking a thread. As presented in Section 2.4.3, the reactive manifesto recommends that systems are message driven and respond asynchronously to ensure loose coupling. The goal for this iteration is to leverage streaming architecture and investigate if it can be leveraged to improve the fault tolerance of a wireless distributed computing platform.

3.4.4.1 Apache Kafka

An established distributed event-streaming platform is Apache Kafka¹⁹. Kafka includes both event transport and storage capabilities through its message brokers. The brokers facilitate the transport of an application's communication by delivering messages published by a producer onto a topic. A consumer can subscribe to the topic and the message broker will deliver events published on the topic to the consumer. Brokers facilitate event storage by using an internal *write-ahead-log*, similar to how database management systems (DBMS) function internally. This allows past events to be replayed by the client. Every event is stored for a configurable amount of time (the default is seven days) after which events are discarded. At the time of writing, Kafka is the only streaming platform to provide *exactly-once* delivery semantics. Delivery semantics refers to the level of guarantees

¹⁹<https://kafka.apache.org>

that the message broker presents when messages are delivered to consumers. The three delivery semantics supported by Apache Kafka are summarised in Table 3.3.

Delivery semantics	Description
At-most-once	guarantees that a message will never be delivered more than once, but does not guarantee that every message will be delivered.
At-least-once	guarantees that a message will always be delivered, but it can be delivered more than once.
Exactly-once	guarantees messages are always delivered but not more than once. This mode incurs a 15-30% throughput penalty in Apache Kafka [128].

Table 3.3 Delivery semantics available in Apache Kafka [129].

The support for *exactly-once* delivery semantics makes Kafka very attractive to distributed transactional systems in which a duplicate event is likely to corrupt the application state. Kafka depends on another Apache software tool called *Zookeeper* to manage cluster metadata and perform other functions including [130]:

- leader election on stream topics and topic partitions;
- the management of service discovery;
- announcing topic additions and removals to Kafka brokers; and
- distributing topology changes to Kafka brokers.

Fault tolerance is one of the key capabilities of streaming platforms [131]. Unfortunately, one of the first trade-offs encountered with this sophistication is configuration complexity. Configuring a Kafka cluster proved to be more complex than any previous iteration. When creating these clusters on the hardware platform of Raspberry Pi's, it proved possible, but not ideal. The major roadblock was the memory demands of Kafka clusters and a Zookeeper quorum. The memory demands were overcome using environment variables that limit the usage of Kafka and Zookeeper below their respective defaults, as presented in Appendix B.2.1.

With a functional Kafka cluster running on the mesh-connected Raspberry Pi's, the drone simulation application developed to use remote procedure calls in the previous iteration needed to be adapted to use the streaming architecture. Most of the desired logic was already developed in JavaScript for the previous iteration, so the application was ported to the new architecture using the *node-rdkafka* library (refer to Section B.2.2 on page 135). Figure 3.15 on the following page presents the high-level architecture for this iteration and Table 3.4 on the following page summarises its advantages over the previous RPC implementation.

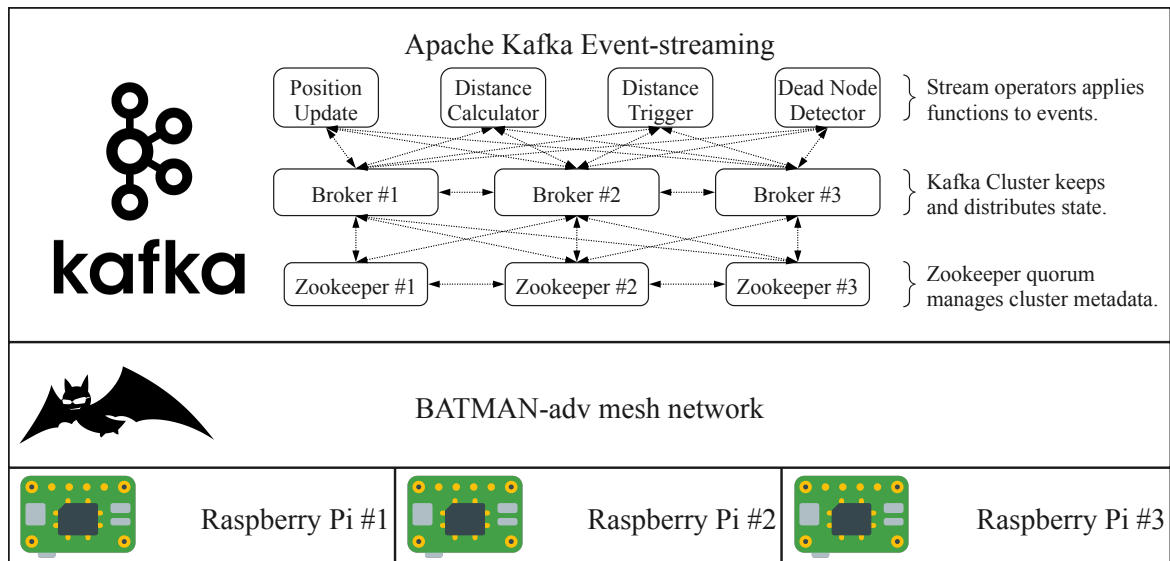


Figure 3.15 High-level architecture for the Apache Kafka implementation.

Advantage	Description
Decoupling	Instead of each server exposing all the functionality, each function could be developed as a separate stream processing application (stream processor). This removed the tight coupling between functions that do not depend on each other.
Simplified networking	Each stream processor only needed configuration for one of the so-called bootstrap servers (Kafka brokers). Once connected to this broker, routing to other brokers in the cluster is handled automatically, so the operator is effectively connected to the entire cluster.
Simpler code	Each of the stream processors is significantly simpler than the RPC server/client implementation. It only had to connect to one of the available brokers, register as a consumer to a specific topic and run its function when a new event is received. Optionally, it could also register as a producer to a different topic and pass the result of its processing to that topic. In this scheme, the complexity resides in the Kafka/Zookeeper configuration, not the application code.
Independent scaling	Each function could now be scaled independently of the others, for example, the <i>deadNodeDetector</i> function is only needed when a node has disconnected and has not broadcast its position for some time. In contrast, the <i>distanceCalculator</i> function is needed every time any of the nodes update their position. By separating these functions into two independent stream processing applications, it is possible to run one instance of the <i>deadNodeDetector</i> application and three instances of the <i>distanceCalculator</i> application to deal with the load.

Table 3.4 Advantages of the streaming implementation over the RPC implementation.

Kafka brokers include many configuration options to control its operation. The configuration

options set to enhance reliability include the replication factor, leader election scheme, required event acknowledgements and the minimum number of event acknowledgements. Their description and use are included in Appendix B.2.1. These configuration options were investigated, but represent a very small portion of the number of configuration parameters available. The implementation of the Kafka cluster provided a significantly more fault-tolerant platform than any of the previous iterations. This fault tolerance introduced more complexity, but also incurred a very high cost in memory requirements: the need to dedicate more than 50% of a node's available memory before any event processing can be done. Because of the complexity and memory usage, it was decided to investigate alternative event-streaming platforms.

3.4.4.2 NATS Streaming

The Cloud Native Computing Foundation presents an interactive landscape of technologies grouped by class and function [132]. At the time of writing, the NATS [133] project is in the CNCF “*incubating*” phase under the Streaming and Messaging category. NATS showed potential for this application as it has a stated focus on providing messaging capability on platforms, including mobile applications and IoT - environments in which memory and processing power are low.

NATS is described as a high-performance, lightweight and secure messaging system. The core NATS server focuses on simplicity and performance first, providing a so-called *fire-and-forget* messaging system. Unlike Kafka that uniquely offers *exactly-once* delivery semantics, NATS only offers *at-least-once* and *at-most-once* delivery semantics. This means that if a consumer is not listening on a topic, or is down when the message is sent, the message will not be received - the same delivery semantics provided by TCP.

In the pursuit of building a fault-tolerant distributed computing platform, a *fire-and-forget* messaging system does not satisfy the reliability requirements. However, a separate NATS Streaming server is available which extends the functionality of the NATS platform. Some of the relevant capabilities added are presented in Table 3.5 on the following page.

This extra functionality provided by NATS Streaming delivers practical feature parity with Apache Kafka as the base for the distributed computing platform. Once again, the drone simulation application was adapted to the new platform, but it required very little adaptation as both Kafka and NATS share the streaming architecture paradigm and the same functions were applied to events. Application functions were again exposed as independent stream processors.

Feature	Description
Event persistence	Events are stored using either in-memory storage, flat files or a database.
At-least-once	ALO delivery semantics leverage the event persistence to allow producers to retransmit messages to eligible consumers as needed.
Message playback	Event persistence also allows consumers to access historical events for a topic, e.g. from the earliest available event or the last 30 seconds' events.
Durable subscriptions	Consumers can now specify a durable name that will survive client restarts. The NATS Streaming server tracks which messages have been acknowledged so that when a client restarts, it can simply resume from the first event it has not received.
Dedicated fault-tolerance mode	By dedicating a single node as the active server with an exclusive lock for event storage and dedicating all others as standby servers, NATS Streaming can compromise on high availability to promote fault tolerance. Standby servers do not interact with stream operators; they are standing by to take over in the event of the active server failing. Note that fault-tolerant mode cannot be used in conjunction with the clustering mode that promotes high availability.

Table 3.5 Capabilities added by NATS streaming over NATS [134].

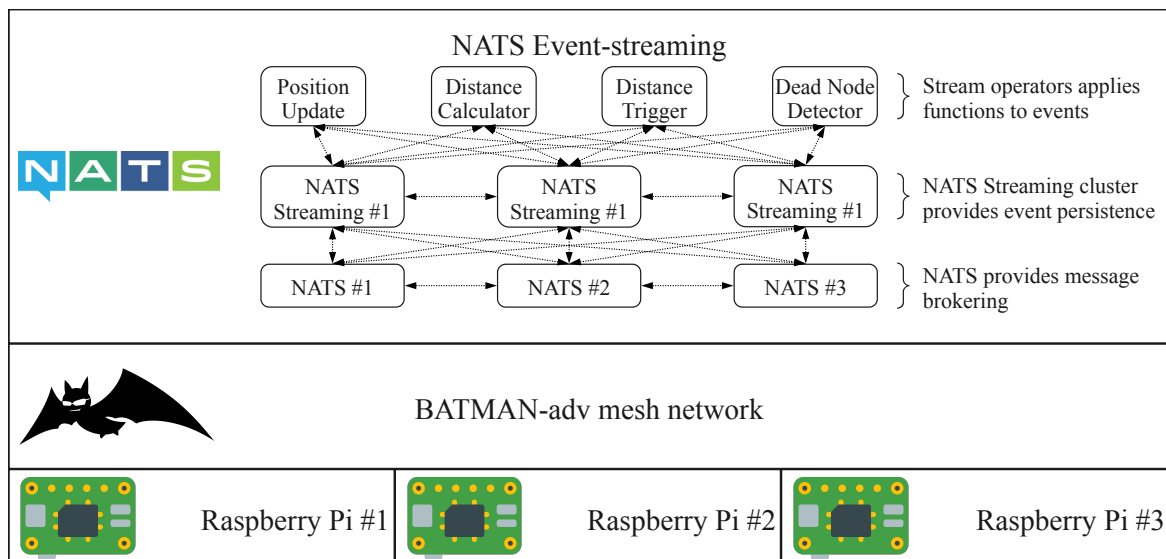


Figure 3.16 High-level architecture for the WebSocket implementation.

The uses of Kafka and NATS Streaming are very similar, as is evident from their high-level architectures presented in Figures 3.15 to 3.16 on pages 71–73, but their implementation and focus are different. One notable difference from a developer's perspective is that unlike Kafka, NATS has *first-class-citizen* support for many languages, not just Java and Kotlin. The NATS and NATS streaming servers are also distributable as single binaries for Linux, Mac OSX and Windows. Finally,

it natively supports ARMv6, ARMv7 and ARM64 directly with its releases. Apache Kafka was compatible with the ARMv7 architecture of the Raspberry Pi due to it running on the JVM, but NATS.io releases compiled binaries for this and other processor architectures.

```
Starting NATS Streams server: drone-simulation-cluster
[1] 2019/07/16 16:23:48.295387 [INF] STREAM: Starting nats-streaming-server[drone-simulation-cluster] version 0.15.1
[1] 2019/07/16 16:23:48.295428 [INF] STREAM: ServerID: SY6h8uHs1t0q9DglvPtHnF
[1] 2019/07/16 16:23:48.295432 [INF] STREAM: Go version: go1.11.10
[1] 2019/07/16 16:23:48.295437 [INF] STREAM: Git commit: [8e6aa7c]
[1] 2019/07/16 16:23:49.089190 [INF] STREAM: Recovering the state...
[1] 2019/07/16 16:23:49.089202 [INF] STREAM: No recovered state
[1] 2019/07/16 16:23:49.339578 [INF] STREAM: Message store is MEMORY
[1] 2019/07/16 16:23:49.339608 [INF] STREAM: ----- Store Limits -----
[1] 2019/07/16 16:23:49.339612 [INF] STREAM: Channels:                100 *
[1] 2019/07/16 16:23:49.339614 [INF] STREAM: ----- Channels Limits -----
[1] 2019/07/16 16:23:49.339617 [INF] STREAM: Subscriptions:           1000 *
[1] 2019/07/16 16:23:49.339619 [INF] STREAM: Messages                : 1000000 *
[1] 2019/07/16 16:23:49.339622 [INF] STREAM: Bytes                   : 976.56 MB *
[1] 2019/07/16 16:23:49.339624 [INF] STREAM: Age                     : unlimited *
[1] 2019/07/16 16:23:49.339627 [INF] STREAM: Inactivity              : unlimited *
[1] 2019/07/16 16:23:49.339629 [INF] STREAM: -----
[1] 2019/07/16 16:23:50.186407 [INF] STREAM: Channel "collision-avoidance-warning" has been created
[1] 2019/07/16 16:23:50.186569 [INF] STREAM: Channel "signal-loss-warning" has been created
[1] 2019/07/16 16:23:50.236574 [INF] STREAM: Channel "positionUpdates" has been created
[1] 2019/07/16 16:23:52.190486 [INF] STREAM: Channel "deadNodeAlert" has been created
[1] 2019/07/16 16:23:52.202873 [INF] STREAM: Channel "distanceUpdates" has been created
```

Figure 3.17 NATS Streams server dedicated to the drone swarm simulation.

Figure 3.17 presents the NATS Streaming server during runtime. During the drone swarm application running, which is presented in Figure 3.18 on the following page, the server uses only 6.82 MB of memory and 6-8% of CPU time on one thread. The NATS Core server on which it depended was storing events in memory, and therefore its memory usage grew as long as events were published. As presented in Figure 3.17, the default configuration would allow a channel to use up to 978.56 MB of in-memory storage or up to one million messages, whichever is filled first. Beyond these thresholds, old messages would be dropped to make space for new ones. The default configuration would not be ideal for the platform, but several options are available as command-line parameters for managing these limits. The options that were used are summarised in Table B.1 on page 136.

As shown in the Kafka implementation, the Raspberry Pi computers' most notable limitation is its low amount of RAM (1024 MB). If that does not provide enough in-memory storage for the number of events needed, the storage should be changed to a file or SQL database. Depending on the application requirements, reducing the maximum total size or the maximum number of stored events is also a viable option.

In the example presented in Figure 3.17, each of the three nodes was updating its position every 50 ms (~20 events/s). Increasing this broadcast rate to every 1 ms (~1000 events/s) resulted in a minor increase in memory usage, from 6.82 to 7.39 MB. However, CPU usage increased more significantly from 6-8% to 42-48%. While further investigation into the resource usage of NATS Streams would be worthwhile, it is immediately evident that this implementation required far less memory than the Kafka implementation, while providing equivalent functionality for this application.

In order to evaluate the NATS Streaming implementation, the application was deployed on the

```

pi@pi-1:~$node /network-experiment/nats-streaming/positionUpdate.js
CurrentPosition: 1222, 1208,1463
Movement: Random
NodeID: 8664
Target: None

pi@pi-2:~$node /network-experiment/nats-streaming/positionUpdate.js
CurrentPosition: 1234, 1221,1457
Movement: Moving closer
NodeID: f4a5
Target: { x: 1217, y: 1197, z: 1478 }

pi@pi-3:~$node /network-experiment/nats-streaming/positionUpdate.js
CurrentPosition: 1226, 1205,1460
Movement: Moving further
NodeID: a3fa
Target: { x: 1225, y: 1207, z: 1466 }

pi@pi-1:~$node /network-experiment/nats-streaming/distanceCalculator.js
Action: Sent DISTANCE-UPDATE
OperatorID: ba1e
Distance Node A Node B
9.055 8664 f4a5
14.283 8664 a3fa
14.071 a3fa f4a5

pi@pi-2:~$node /network-experiment/nats-streaming/distanceTrigger.js
Action: Sent SIGNAL-LOSS warning
Detected: f4a5: DIST WARNING
OperatorID: 213b

pi@pi-3:~$node /network-experiment/nats-streaming/deadNodeDetector.js
OperatorID: a475
Node 8664 last update: 116 ms ago
Node a3fa last update: 25 ms ago
Node f4a5 last update: 42 ms ago

```

Figure 3.18 NATS Streams implementation of the drone swarm simulation.

three available Raspberry Pi 3B+ nodes. Six SSH sessions are presented in Figure 3.18 and each session represents stream operator instances and are described in Table 3.6 on the following page.

A logical overview of the streaming application is presented in Figure 3.19 on the following page. Note that this figure presents the NATS-specific implementation, but the organisation of stream processors and topics also applied to the previous Kafka implementation. In this figure, circles depict stream processing applications (stream operators) while diamonds represent topics on which these operators send and receive events. The topics are abstractions of communication channels presented by the NATS Streaming server. Arrows between these operators and topics represent the logical flow of events and construct a feedback loop which maintains the desired application state. The left purple block represents the NATS streaming servers, and the right purple block represents the standard NATS server on which it depends.

Session	Stream processor	Description
Session 1-3	<i>positionUpdate</i> on <i>pi-1</i> , <i>pi-2</i> and <i>pi-3</i> .	This operator represents a “drone” and is the initial <i>producer</i> into the streaming application, periodically publishing generated position updates. This application also receives warning events and moves closer to or away from other nodes in response.
Session 4	<i>distanceCalculator</i> on <i>pi-1</i> .	This operator receives events from <i>positionUpdate</i> operators and calculates the relative distances between them. It then publishes this information onto a specified topic. The operator also receives events from the <i>deadNodeDetector</i> operator to prevent it from calculating relative distances to dead nodes.
Session 5	<i>distanceTrigger</i> on <i>pi-2</i> .	This operator receives events from the <i>distanceCalculator</i> operator and checks whether any of the reported distances are outside a specified range. If a pair of nodes is either too close or too far from each other, the operator generates a warning event to be consumed by these <i>positionUpdate</i> operators.
Session 6	<i>deadNodeDetector</i> on <i>pi-3</i> .	This operator receives events from the <i>positionUpdate</i> operators and records the time at which it was received. It then periodically checks whether any node has not sent an application update in a specified time, e.g. five seconds. If a node crosses this threshold, this application will assume the node has disconnected or failed, and produce a dead node warning event to be consumed by the <i>distanceCalculator</i> operator.

Table 3.6 Description of sessions presented in Figure 3.18 on the previous page. The sessions are grouped by which streams operator is running on them. Figure 3.19 shows the combination of these operators to form the distributed fault-tolerant application.

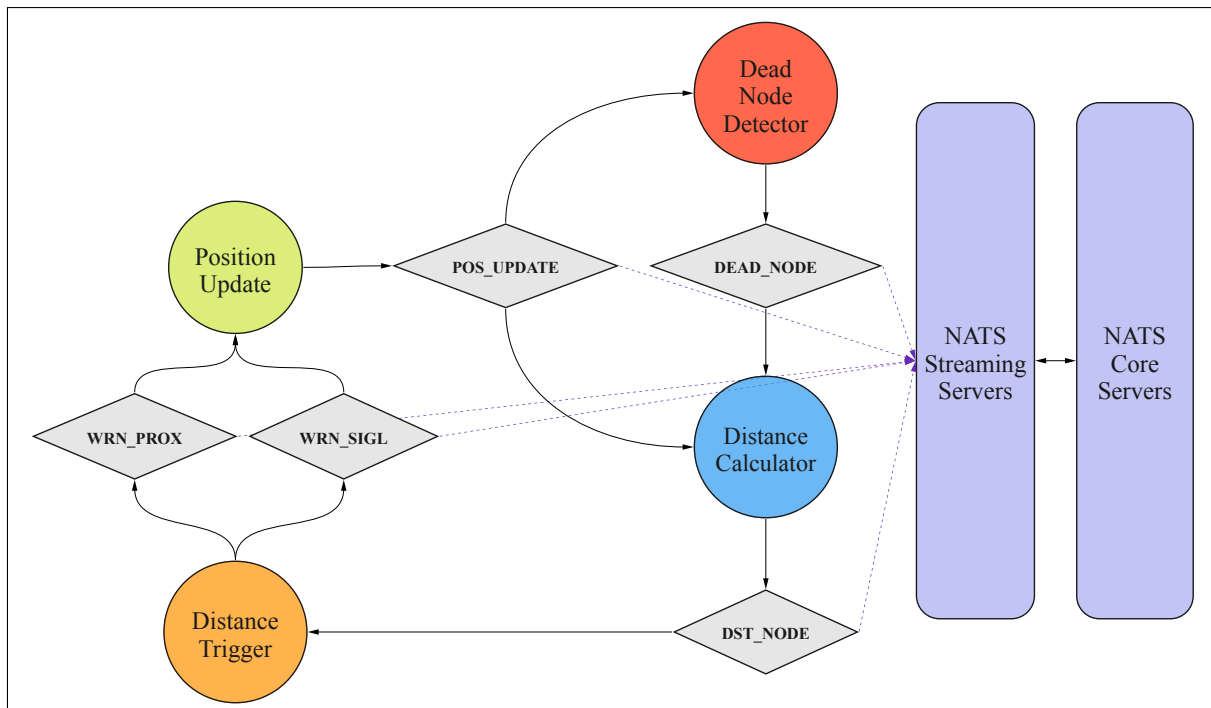


Figure 3.19 Streaming architecture implementation of the drone swarm simulation.

This implementation provided all the fault-tolerance benefits of the Kafka implementation, but introduced less complexity and required a fraction of the memory. NATS Streaming was therefore chosen as the solution with which to build a distributed computing platform for this research. With the application design complete, the focus shifted to the deployment and runtime of the application. Besides designing the application to promote fault tolerance, could it be enhanced further by leveraging modern deployment techniques? Section 3.5 presents an overview of containerisation, an application deployment technique popular with distributed microservice applications. Containerisation is introduced in Section 2.4.4 on page 45.

3.5 Containerisation of the platform

In all the iterations presented in Section 3.4, the application was running directly on the chosen hardware platform: three Raspberry Pi 3B+ SBCs. The necessary software packages needed to be installed; typically using the *apt* utility to pull packages from package repositories. In some cases, such as MPICH, the software had to be installed by compiling executable binaries from the source code. This process takes a considerable amount of time on a less powerful device like the Raspberry Pi. This approach proved quite tedious because every time changes are made to application code, application configuration or the runtime environment, these changes need to be replicated or adapted across every machine. This needs to be done without introducing inconsistencies in configuration that can cause crash faults (see Figure 2.2 on page 17). Beyond configuring the application for each node, the application itself is implemented as a set of microservices. Manually performing the configuration of this interdependent set of components across several nodes becomes a deployment challenge that should be automated to reduce the risk of configuration faults that lead to application crashes (see Section 2.1.1).

Docker²⁰ is a software platform that allows the binding of software packages, dependencies and configuration into a single, easily distributable container image. When dealing with multiple nodes that need the same software set, such as the MPICH runtime environment, containerisation can be of enormous value in time savings alone. This approach was useful in the attempt to containerise the cluster developed in the first iteration.

Besides simplifying deployment, containerisation can also be leveraged to produce a much more fault-tolerant distributed system when deploying components in an orchestrated environment [93], [94]. In this section, the process followed in leveraging container orchestration to increase the fault tolerance of the drone swarm application developed in Section 3.4 will be presented. In this research, containerisation produced several benefits summarised in Table 3.7 on the following page.

The initial benefits of containerisation centred around convenience, e.g. eliminating duplicate work. However, containerisation is extremely popular for a very different reason: container orchestration. Container orchestration is introduced in Section 2.4.4. This approach to distributed systems design

²⁰<https://www.docker.com/>

Benefit	Description
Time savings	Long-running processes, such as compiling software from source, only need to be performed once, and the result of the compilation is easily distributed among all nodes.
Node consistency	When managing multiple systems, keeping software versions and configurations consistent can be a challenge. By packaging all this complexity into a single unit, containerisation makes it trivial.
Simple scaling	If a new node needs to be added to the platform without containerisation, it has to be set up from scratch, and it was challenging to ensure that it was running the same versions of packages with the same configuration. If the application is containerised, the process is as simple as pulling the containers from a registry and running them with the correct configuration.
Replicated services	Application components could be deployed as a service with a defined replica count, e.g. three instances spread over a variable number of nodes. Alternatively, the service can be deployed as a global service - ensuring each node is running one instance of the component. In this scheme, when a node is added, it will automatically fetch the defined image and start up an instance of it.

Table 3.7 Benefits of containerisation in distributed application development.

naturally fits with the microservice architecture presented in Section 2.4.2. Building the distributed computing platform as an orchestrated collection of loosely-coupled components packaged as easily distributable containers yields additional benefits. Most notably, container orchestration can be leveraged to increase fault tolerance by replication as presented in Section 2.1.3.2.

Two well-known container orchestrators include Google’s *Kubernetes* (abbreviated to *k8s*) and Docker’s Swarm orchestrator. Figure 3.20 on the following page shows that, based on Google search data from the United States, there is significantly more interest in the Kubernetes orchestration tool than in Docker Swarm. Despite this, configuring Kubernetes on the Raspberry Pi proved very complex in comparison to Docker Swarm. The initialisation of the k8s cluster could not be completed. This step is expected to take about 15 minutes, but after an hour, it still had not completed. The experience matches an issue reported on the *kubeadm* GitHub page²¹. After multiple failed attempts, it was decided to investigate the use of the less popular *Docker Swarm*.

Docker Swarm is included with the standard Docker installation and provided a much simpler setup experience than k8s. Initialisation of a Swarm simply involved a single initialisation command on a leader node, which will generate a token to be used on worker nodes. Assuming a functional network, running this command on other nodes creates a functional orchestrated environment in minutes. The installation of Docker and the configuration of Docker Swarm is presented in Section B.1.2.

Once a functional Swarm is created, it is possible to create so-called *overlay* networks. An overlay

²¹<https://github.com/kubernetes/kubeadm/issues/413>

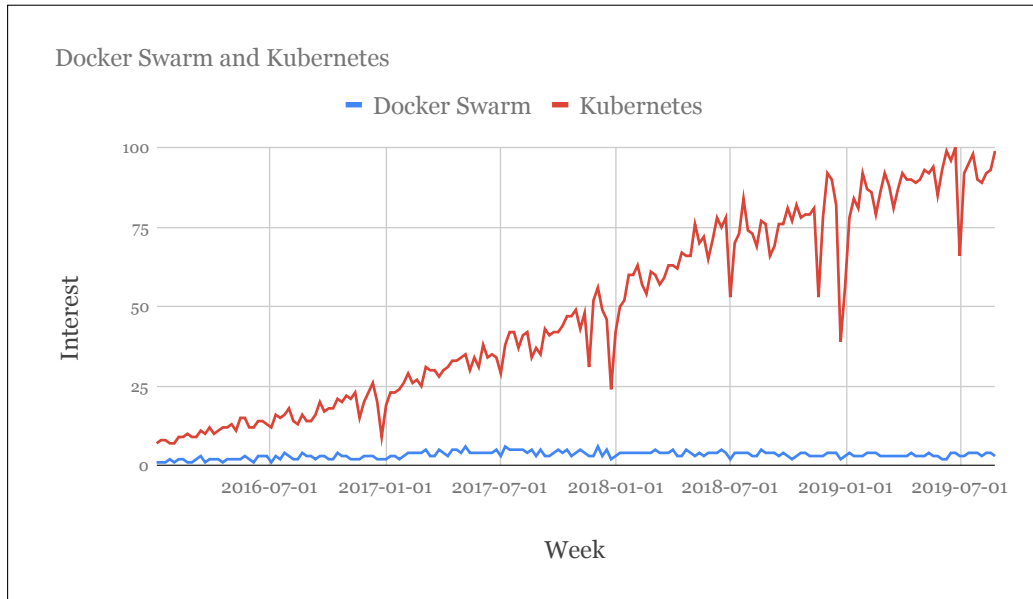


Figure 3.20 Comparison of interest between Kubernetes and Docker Swarm based on Google Trends data.

network is a Virtual Extensible LAN (VXLAN) network used by the containers running in the swarm. The overlay network abstracts away the complexities of the physical network to the extent that the containers can communicate using simple IPv4 addresses or internal DNS records. Overlay networks are well-suited to simplifying networking in complex environments. Examples of such environments include multi-cloud or hybrid-cloud environments where the nodes in the swarm might not be in the same data centre, country or even continent. In the artefact of the research, the overlay network will abstract away the routing of the IEEE 802.11s mesh network provided by the *bat0* virtual network interface.

In the Docker Swarm context, the deployment unit of an application that comprises multiple different containers is called a *stack*. A minimal stack could include a defined network and two defined services. A service is characterised by one or more instances of a container running a specified image. This configuration is built into a structured YAML file which is designed to be a “*human-friendly data serialisation standard*” [135]. The *docker-compose* tool was used during the development of the stack and allowed the distributed application to run in a non-orchestrated environment, e.g. a single development machine. Once the Swarm orchestrator was functional, the application was deployed with the command presented in Script 3.1 on the following page.

The drone swarm simulation application was successfully packaged as a set of containerised microservices and deployed in the orchestrated Docker Swarm environment on nodes connected using the 802.11s mesh network. The *docker-compose* script used for this deployment is presented in Section B.2.4.

```

1 $ docker stack deploy --compose-file docker-compose.yml drone-sim
2 #result:
3 Creating network drone-sim_main
4 Creating service drone-sim_nats-2
5 Creating service drone-sim_nats-processor-position-update-1
6 Creating service drone-sim_nats-processor-distance-calc-1
7 Creating service drone-sim_nats-1
8 Creating service drone-sim_nats-processor-position-update-3
9 Creating service drone-sim_nats-3
10 Creating service drone-sim_nats-streaming-1
11 Creating service drone-sim_nats-processor-deadnode-detector-1
12 Creating service drone-sim_nats-processor-position-update-2
13 Creating service drone-sim_nats-processor-distance-trigger-1
14
15 $ docker stats --format "table {{.Name}}\t{{.CPUPerc}}" --no-stream
16 #result:
17 NAME                                                                    CPU %
18 drone-sim_nats-3.1.ypr6w2y8o6amj0epfcz5tolv6                          1.78%
19 drone-sim_nats-processor-distance-trigger-1.1.mov7i5hbhtkgb3p2mozxmuo15 0.06%
20 drone-sim_nats-processor-position-update-2.1.moax93oczmh7u4ubm0evy86qe 0.03%
21 drone-sim_nats-processor-deadnode-detector-1.1.z3ligb6aj1c2ph5trlepictpu 0.00%
22 drone-sim_nats-processor-position-update-3.1.tojusptgbf9mt2qpk6hbh35rh 0.07%
23 drone-sim_nats-processor-distance-calc-1.1.n78twclyf6j595fv12gck6p3s 0.04%
24 drone-sim_nats-processor-position-update-1.1.qf6dj5utimcxxp13d25mi5wrp 0.01%

```

Script 3.1 Result of deploying the Docker Stack to provision the application.

3.6 Summary

The goal in this chapter was to create a fault-tolerant platform to support wireless distributed computing. The previous section was concluded with a functional containerised distributed computing application, running in an orchestrated environment on Raspberry Pi nodes connected via a mesh network. This deployment represented the combination of efforts at multiple levels of the architecture to enhance fault tolerance.

At the networking level, the mesh network eliminated a single point of failure by enabling redundant routing paths. At the application level, the use of the NATS Streaming server allowed interdependent streaming processors to enhance fault tolerance by decoupling data and processing. At the deployment level, the application was successfully containerised and deployed in an orchestrated environment with abstracted networking. The final iteration of the platform demonstrated its fault tolerance and would continue functioning in several challenging scenarios:

1. When any of the individual Raspberry Pi nodes failed or were disconnected, the platform would continue functioning with the remaining nodes.
2. By flattening the hierarchy and making every node a Swarm Manager, there is no reliance on any single manager node to manage the cluster state. Instead, the global cluster state is managed by Docker Swarm managers using the Raft Consensus Algorithm [136].

3. NATS Streaming provides event persistence in addition to the event transport provided by NATS. Their combination, presented in Figure 3.21 on the following page, ensures that when any of the client applications fail, the container that ran them can be re-provisioned on any available host. Once the service started up and rejoined the NATS cluster, clients will automatically start sending and receiving events to/from the newly revived broker.
4. The overlay network driver included in Docker Swarm mode ensures that network routing to containers that failed on one host and was re-provisioned on another remains functional. This is a core function of the service discovery features available in Docker Swarm.

Layer	Section	Description
Hardware	3.2	Three Raspberry Pi 3B+ single-board computers that will run the chosen software of the platform. This choice was driven by support for the platform and their low cost.
Network	3.3	IEEE 802.11s mesh network constructed with BATMAN-adv as the OSI layer-2 (refer to Section 2.2.1) routing algorithm. This design choice enhances fault tolerance by eliminating the need for a single access point. If there is no single-hop transmission path available between two nodes, an available multi-hop transmission path could enable communication.
Application	3.4	The software of the distributed computing platform is designed as a set of event-streaming microservices that communicate via the NATS Streaming solution. This choice enhances fault tolerance by decoupling application data (state) from processing.
Deployment	3.5	Both the streaming platform (NATS) and streaming application (operators) are deployed as a collection of containerised services in a Docker Swarm orchestrated environment. This approach enhances fault tolerance by enabling and simplifying replication. Due to the application being implemented as an interdependent set of replicated microservices, individual containers can fail without causing the application to fail catastrophically and present a system error to the user. The Swarm orchestrator can also provision new containers to replace failed containers automatically.

Table 3.8 Summary of the layered platform architecture. References to the section that informed each layer’s design choices are provided with descriptions of the layer and how it contributes to the fault tolerance of the platform as a whole.

This chapter concludes with the final architecture of the artefact. Its conceptual levels are summarised in Table 3.8 with a description of the efforts that promote fault tolerance. The architecture is presented in Figure 3.21 on the following page and is the result of a practical application of systems architecture. In this chapter, steps 3 and 4 of the DSR approach as presented in Section 1.6.1 were completed. The artefact’s final design was produced and each iteration demonstrated increased fault tolerance when evaluated using the illustrative scenario presented in Section 3.1. In reference to the research approach presented in Section 1.6, the artefact created in this chapter simultaneously

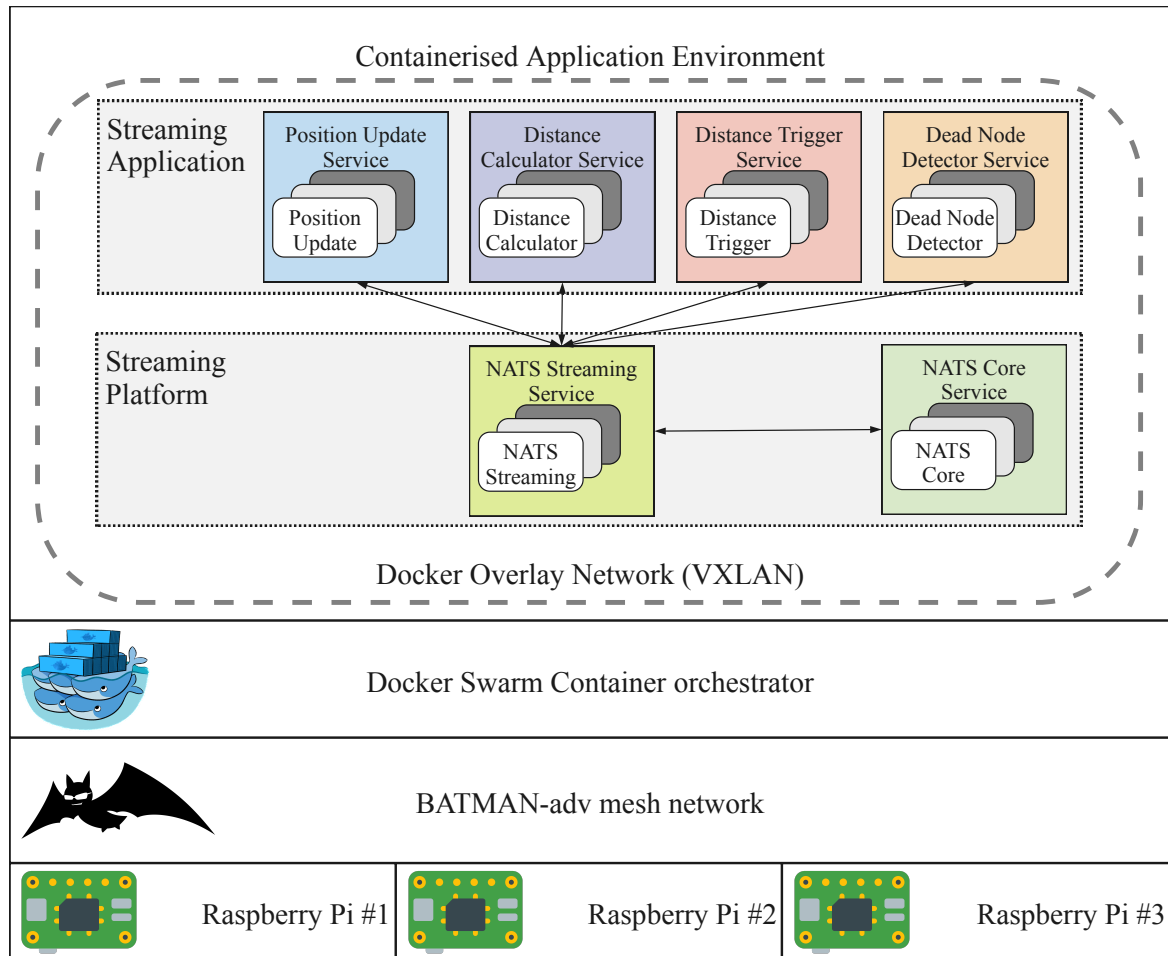


Figure 3.21 Architecture diagram of the fault-tolerant wireless distributed computing platform.

answered knowledge questions and solved design problems. In the next chapter, the artefact produced will be evaluated using technical experiments, presented in Section 1.6.3.

Chapter 4

Artefact Evaluation

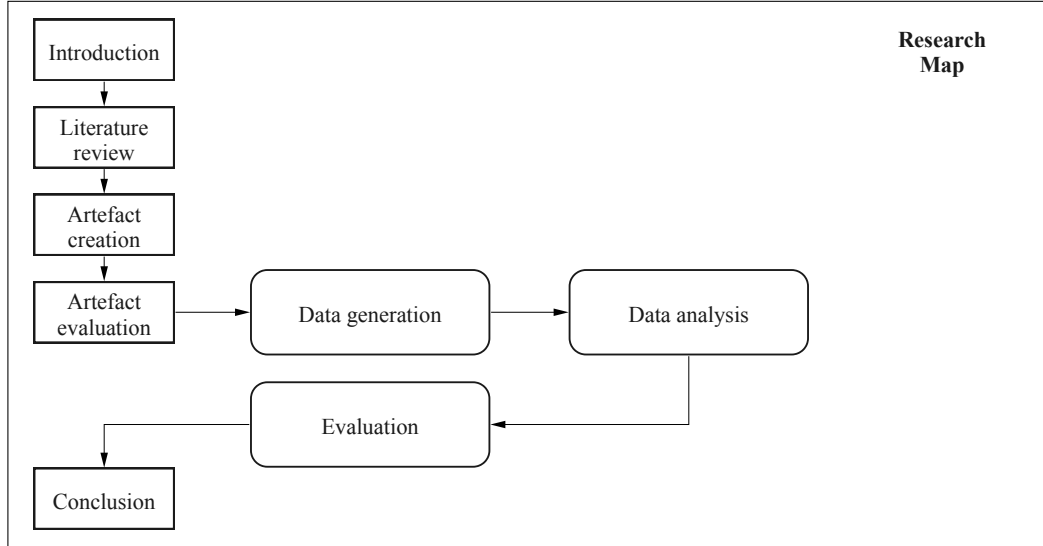


Figure 4.1 Research map extract for Chapter 4. The complete research map is presented on page 1.

In this chapter, the evaluation of the artefact developed in Chapter 3 is presented. In terms of the DSR steps presented in Section 1.6.1, evaluation is the fifth step. The artefact developed in the previous chapter represents the two chosen outputs of the DSR approach presented in Section 1.6: the instantiation and the method.

The artefact development process, presented in Chapter 3 followed an iterative development process driven by an illustrative scenario evaluation method, applied at the end of each iteration. The illustrative scenario was a simple flight control system leveraging wireless distributed computing, presented in Section 3.4.2. Each iteration of the platform aimed to improve upon the fault tolerance demonstrated by its predecessor. These incremental improvements are summarised in Table 3.8 on page 81. This method of evaluation was strictly qualitative and provided insight into the architectural challenges inherent in developing distributed systems. In the next sections, the final iteration of the artefact is evaluated using a second method: technical experiments. This method is applied to produce insight into the performance of the artefact by measuring the relationship between cause and effect.

By manipulating one component of the platform as an independent variable (cause) and measuring a dependent variable (effect), insight into the relationship between the components can be generated. As the performance of a distributed computing platform is heavily reliant on the performance of its communication system [137], [138], an appropriate dependent variable for the experiments is the QoS metrics introduced in Section 2.2.5. These four metrics represent elements of speed and reliability that define a user's (or application's) experience in using the network. Three variables

have been selected as independent variables to be investigated:

- transmission distance;
- network adapter mode; and
- native vs. overlay networking.

Experiments that determine the relationship between these variables and the four QoS metrics are presented in Section 4.1.

4.1 Artefact evaluation using technical experiments

In this section, the artefact is evaluated using technical experiments. The experiments are divided into two sets; the first set of experiments is focused on the networking aspects of the artefact, presented in Section 4.1.1. The second set will focus on the event-streaming aspects of the artefact, presented in Section 4.1.2.

4.1.1 Network experiments

This section is dedicated to presenting the experiments performed to evaluate networking performance in the artefact. Three experiments were performed to achieve this goal:

1. Modelling the relationship between transmission distance and network performance;
2. Comparing the network performance of standard infrastructure wireless networking to wireless mesh networking; and
3. Measuring the networking performance impact of the Docker overlay network.

Experiment procedure: In each experiment, the QoS metrics are measured as the dependent variable, while the transmission distance, network adapter mode or use of a virtual network are manipulated as the independent variable.

The bandwidth, packet loss and jitter metrics were measured using the *iperf3*¹ utility in its server/client setup. Each node started an *iperf3* server process. Depending on the test, clients sent either TCP or UDP packets to the server which reported the results. Network latency was measured using the *fping*² utility which sends ICMP packets with the default size of 56 bytes each second.

4.1.1.1 Experiment 1: The effect of transmission distance on network performance

In this experiment, the typical wireless networking configuration was used: a single wireless access point with multiple clients in IEEE 802.11n mode. The experiment procedure is presented in Figure 4.2 on the following page. In each iteration of the experiment, two nodes are connected to the access point³ at incrementally increased distances. The access point used has a typical transmission

¹<http://iperf.fr>

²<https://fping.org>

³https://www.trendnet.com/products/product-detail?prod=215_TEW-691GR

power of 15 dBm, whereas the client NICs reported a transmission power of 30 dBm. The result of this experiment is presented in Figure 4.3 on the following page.

By measuring the QoS metrics between the AP and the antenna of the NICs over distance increments ranging from 0 to 400 cm, a distance profile was created. Between 0 cm and 50 cm, measurements were taken in 10 cm increments, after which the increment was increased to 50 cm.

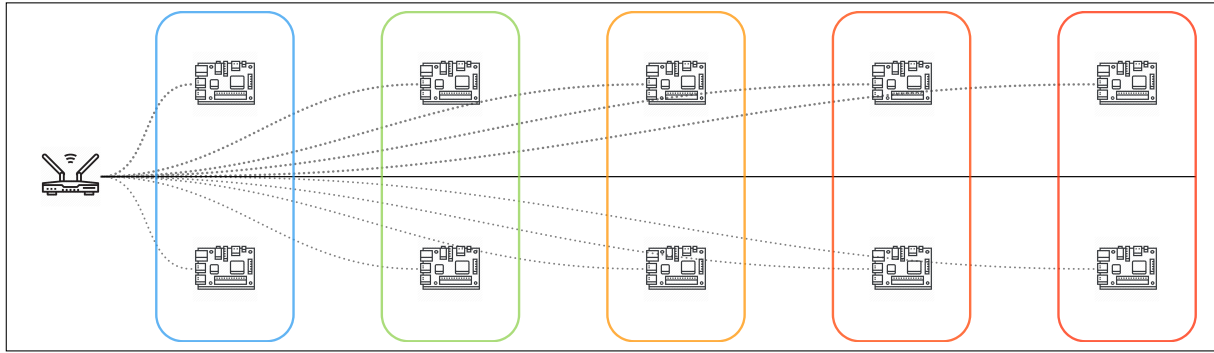


Figure 4.2 Representation of the distance experiment setup. The default ping parameters were used; a standard 56 byte ICMP packet once per second for 30 seconds.

The distance profiles are presented in Figure 4.3 on the following page. Each sub-figure presents a different QoS metric, with the red line indicating the average value (over 30 repetitions) and the standard deviation (σ) value around this value is indicated with shading. A narrow shading indicates a low σ value (more consistent results) while larger shaded areas indicate a higher σ value (less consistent results). The consistency of the results show secondary effects of distance on each measured QoS metric, and it is evident that larger distances generated both worse average measurements and less consistent results.

Reviewing the results presented in Figure 4.3 on the following page, shows that beyond 250 cm, network performance starts to degrade and become less consistent. The results also show that placing the NIC directly next to the AP yields poor results, particularly in the case of bandwidth (Figure 4.3a on the following page) and jitter (Figure 4.3d on the following page). These measurements seem to indicate a signal “dead-zone” when the access point and the network adapter are right next to each other. Figure 4.4 on page 87 illustrates three distinct regions in which radio waves have different characteristics. The distance of one wavelength is given by:

$$\alpha = v/f \quad (4.1)$$

Where:

- α is the wavelength in meters;
- v is wave velocity in m/s; and
- f is the wave frequency in Hz.

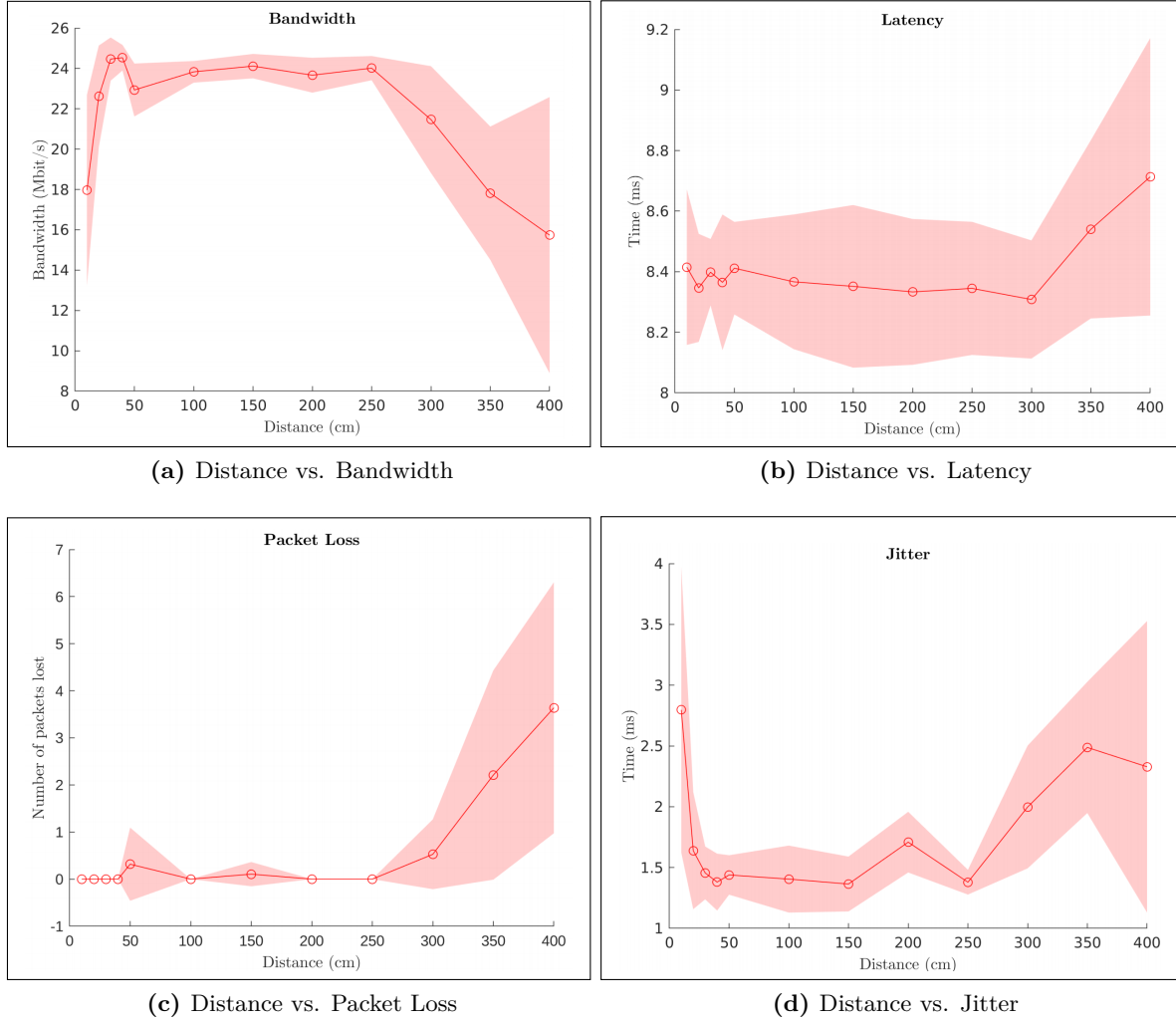


Figure 4.3 Distance profiles of the four Quality of Service metrics.

Using equation 4.1, the wavelength of 2.4 GHz radio α is 12,491 cm. Figure 4.4 on the following page represents both the near-field and far-field regions with the reactive zone (up to $0.159\alpha = 19.866\text{mm}$). Within this zone, the sending and receiving antennas can have capacitive or inductive interaction that would give false results [139]. Beyond 2α ($\pm 25\text{cm}$), the waves are in the far-field region and the results presented in Figure 4.3 are more reliable.

The results of this experiment informed the design of subsequent experiments that used a central access point. In these tests, the network clients were placed at $\sim 150\text{ cm}$ from the access point. Reasonable measures were taken to minimise other sources of signal interference, including:

- Removing or powering down any high-powered devices, such as computers, loudspeakers and printers from the room in which the experiment is conducted;
- Removing all devices that transmit data wirelessly, such as smartphones and Bluetooth peripherals from the room;

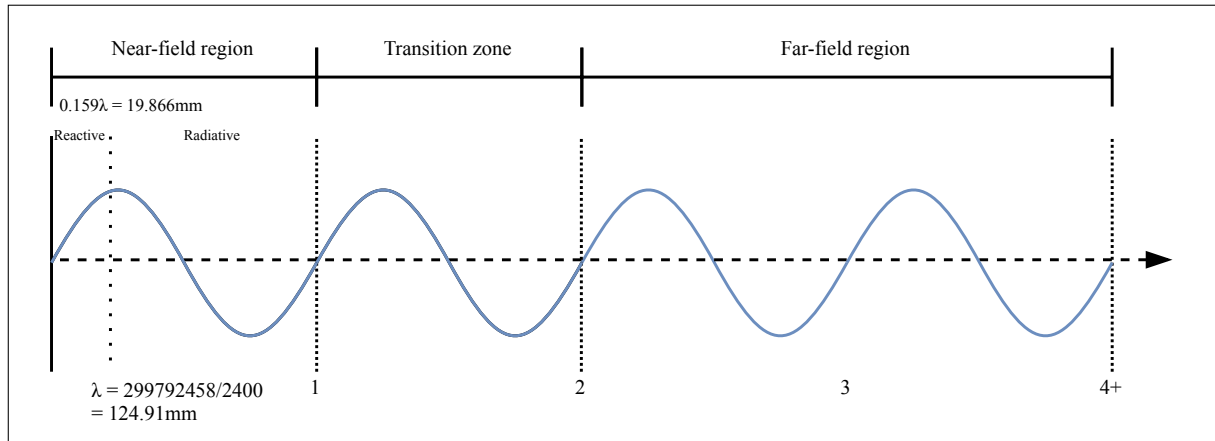


Figure 4.4 Near-field, transitional and far-field regions in wireless communications (Figure adapted from [140]). The distance of one wavelength in 2.4 GHz transmissions, as is typically used in IEEE 802.11n, is approximately 12.5 cm. The reactive zone within the near-field region is indicated at 0.159α .

- Ensuring that only the devices included in the experiment are connected to the network; and
- Using the *wavemon* utility, presented in Figure 4.11 on page 92 to ensure that the adapter has a 100% link quality before starting the experiment.

4.1.1.2 Experiment 2: Comparison between infrastructure and mesh networks

In this section, the second experiment that aimed to compare the network performance of identical networking hardware in two separate modes: infrastructure and mesh-point mode, is presented. Figure 4.5 on the following page presents the two scenarios to compare. On the left is the typical infrastructure mode wireless network used in Section 4.1.1.1: a single access point facilitating communication between all of the nodes connected to it. On the right is a mesh network, without the access point.

For the artefact created in Chapter 3, the mesh network was chosen because it eliminated a single point of failure in the platform: a central communication hub. Comparing the network performance of the two network modes will provide insight into how large a performance penalty, if any, is incurred for the ability to operate the wireless network without an access point. The experiment procedure is identical to that performed in Section 4.1.1.1. In this experiment, the independent variable is the network interface mode. The same utilities (*iperf3* and *fping*) were used to do measurements. The results of the four tests are summarised in Table 4.1 on the following page and presented in Figures 4.6 to 4.9 on pages 89–90. The results are mean values calculated by taking 30 instantaneous measurements over 30 seconds for all three of the node pairs (N1-N2, N1-N3 and N2-N3).

The results summarised in Table 4.1 on the following page indicate that the adapters deliver significantly better performance in mesh mode than in infrastructure mode. In all four metrics, the mesh networking implementation delivered the better average result and in the case of packet

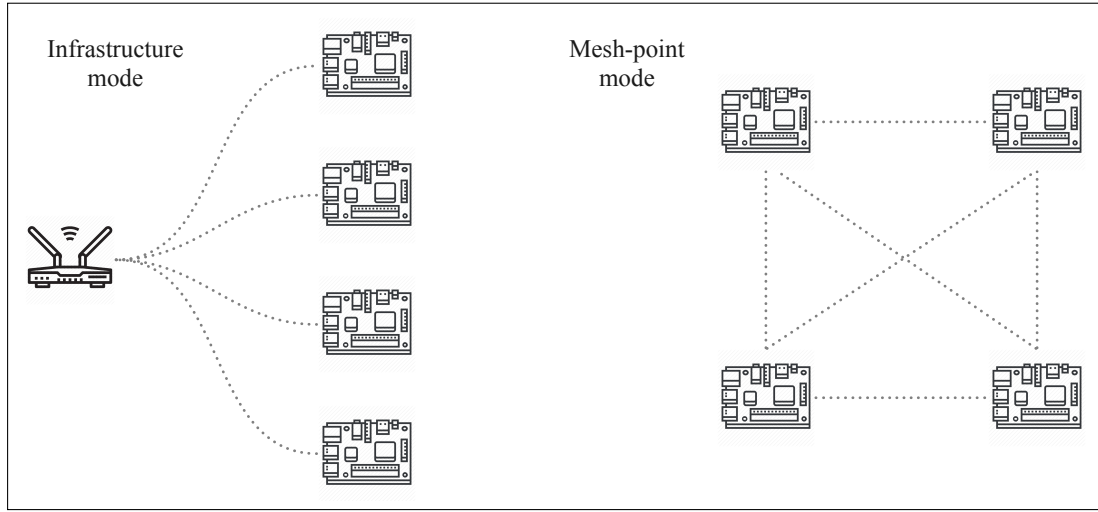


Figure 4.5 Infrastructure vs. mesh-point mode configurations. The design of the platform (presented in Figure 3.21 on page 82) favours the mesh-point network mode for increased fault-tolerance. Note that while this figure presents four nodes, only three nodes were used in the platform.

loss and jitter, delivered more consistent results as well. The packet loss result of 0% for the mesh networking mode was very unexpected, but repeating the test produced the same result.

A plausible theory for the 0% packet loss is related to the way the *batman-adv* mesh-routing protocol functions. *Batman-adv* effectively creates a virtual network switch [141] in which all traffic passing through it is encapsulated in ethernet frames and forwarded to its intended destination. The *iperf3* utility used to measure packet loss transmits UDP datagrams without the reliability associated with TCP such as establishing a reliable transmission, such as establishing a route, applying congestion control, requiring acknowledgements or re-transmitting lost packets. This behaviour explains an average packet loss ratio of 7.8% for the infrastructure mode, which is in keeping with literature [8].

The virtual network created by *batman-adv* implements its own transport characteristics as a routing protocol irrespective of what data packets it encapsulates. Unfortunately, the documentation available for *batman-adv* is implemented as a collaborative wiki and does not provide information to confirm or refute this theory. What is observable from the application level is that the mesh

	Infrastructure		Mesh		
	μ	σ	μ	σ	Note
Bandwidth (Mbit/sec)	3.7140	0.0485	9.0140	0.1267	Higher μ is better
Latency (ms)	3.6537	0.1649	1.7543	0.3776	Lower μ is better
Packet loss (%)	7.8689	0.5577	0.0000	0.0000	Lower μ is better
Jitter (ms)	20.1370	0.4472	12.0632	0.3254	Lower μ is better

Table 4.1 Summary of mean (μ) and standard error (σ) results for the QoS metric measurements comparing infrastructure and mesh networking modes.

network implementation outperforms the infrastructure implementation by a significant margin.

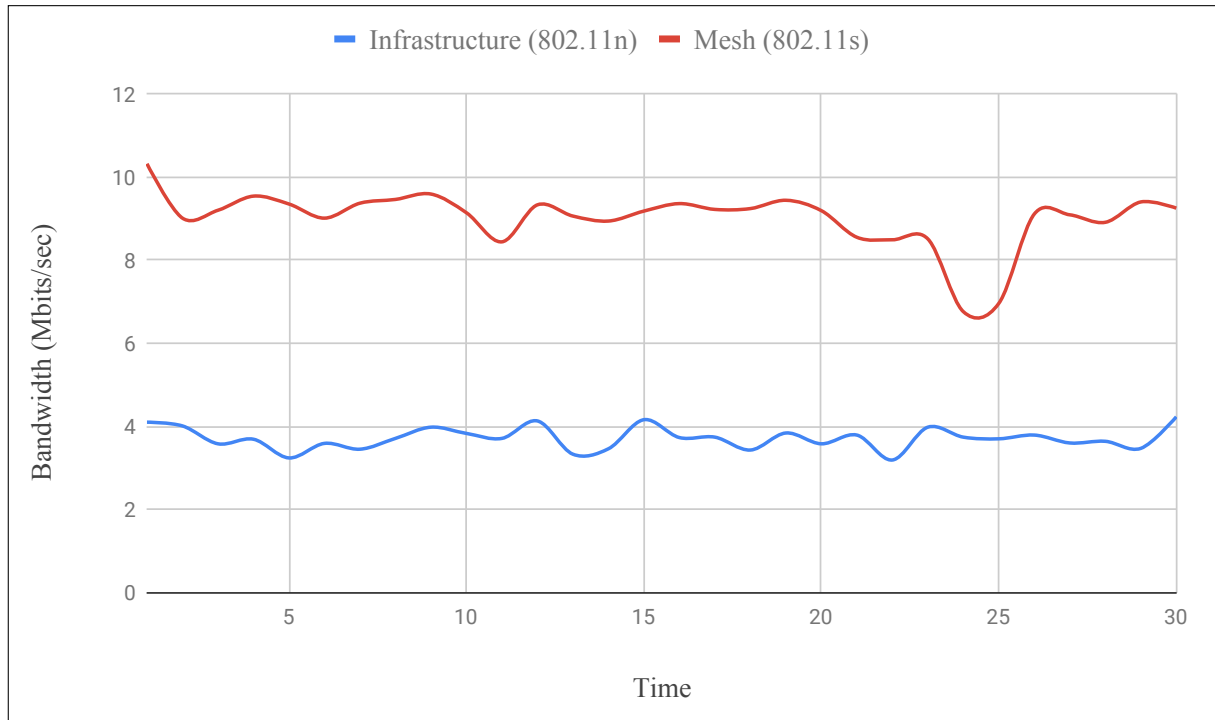


Figure 4.6 Mean bandwidth comparison over 30 seconds between infrastructure and mesh network modes.

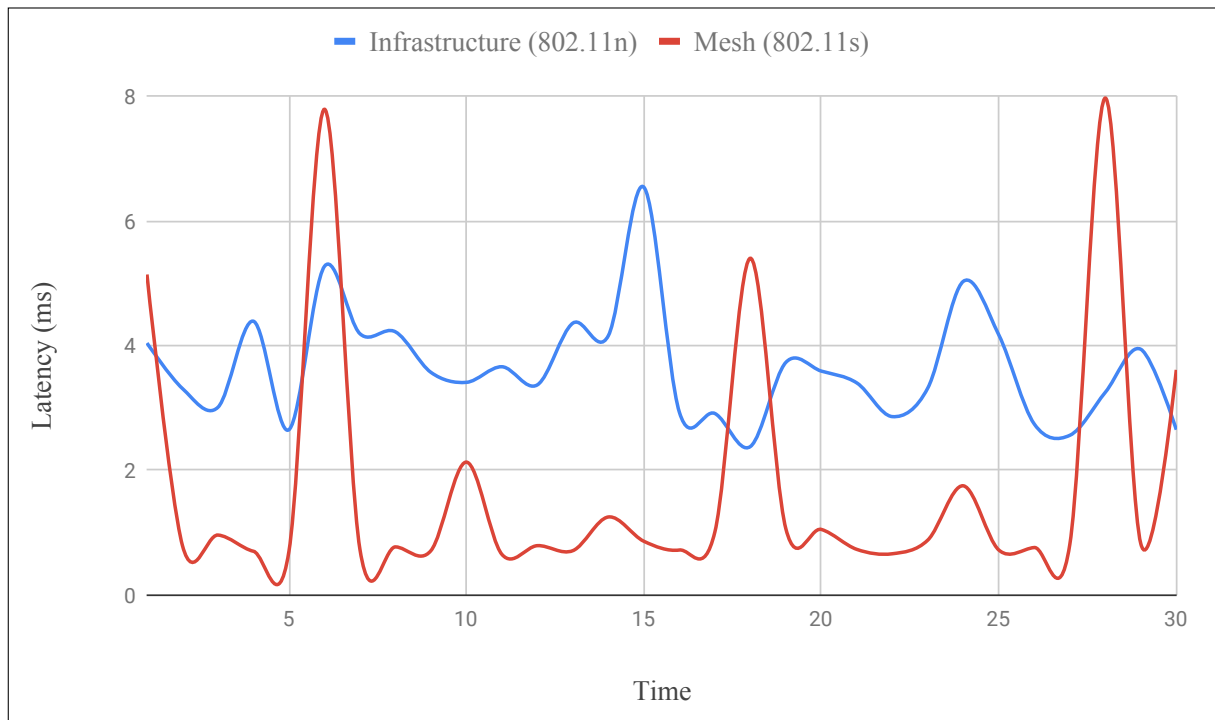


Figure 4.7 Mean latency comparison over 30 seconds between infrastructure and mesh network modes.

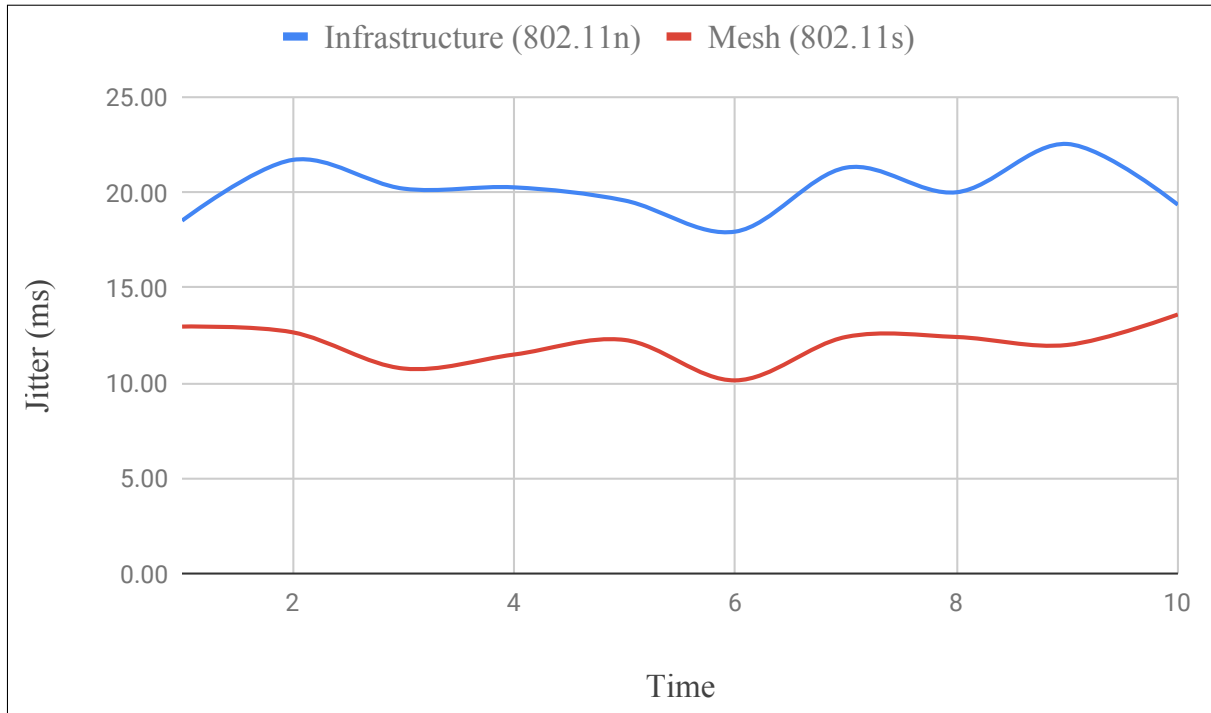


Figure 4.8 Mean jitter comparison over 30 seconds between infrastructure and mesh network modes.

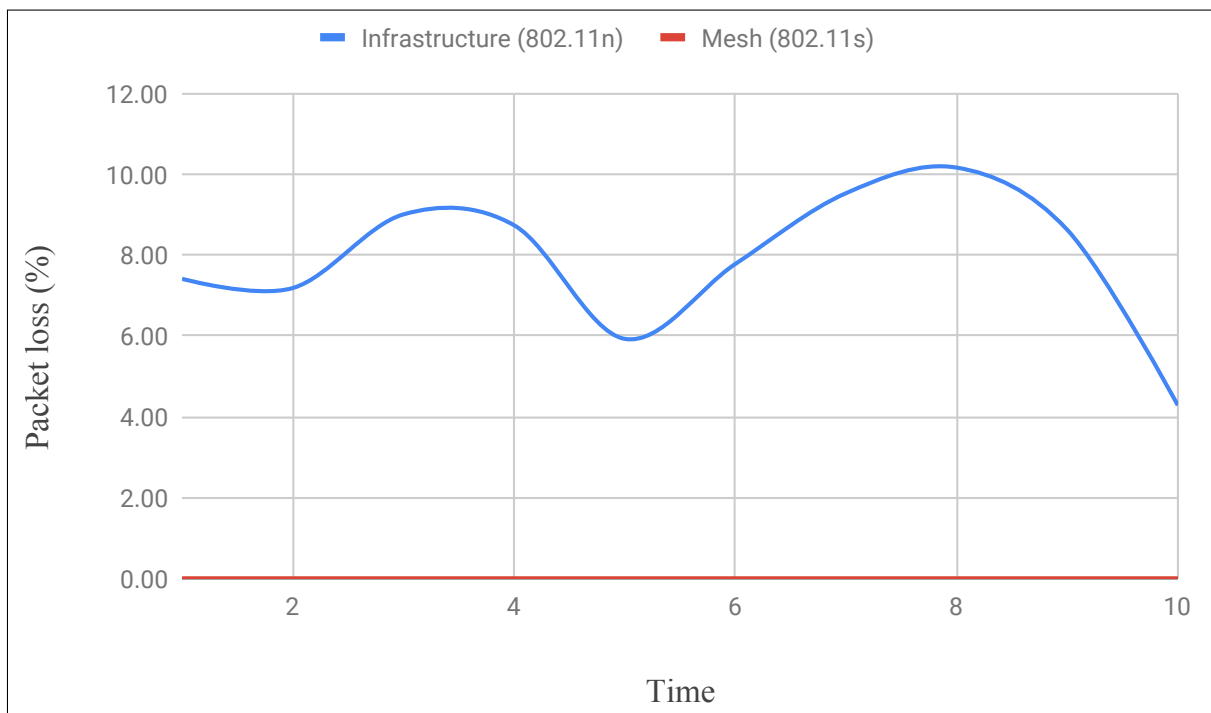


Figure 4.9 Packet loss comparison between infrastructure and mesh network modes.

4.1.1.3 Experiment 3: Measuring the impact of the Docker overlay network

Referring to Figure 3.21 on page 82, there are two distinct networks employed in the artefact: the physical and virtual networks. The physical network refers to the mesh network implemented using *batman-adv* that the three nodes in the artefact use to communicate. This underlying physical network could be replaced with Ethernet, and the infrastructure Wi-Fi network or a mixture, without needing to reconfigure the virtual network: a VXLAN created using the overlay network type in Docker. This virtual network depends on the physical network and is used to facilitate communication between containers that can reside on any of the nodes. In this section, the third experiment is presented.

The aim of the experiment is to quantify the impact of the overlay driver on network performance from the application level. The typical infrastructure mode IEEE 802.11n wireless network was used to isolate any changes of the Docker overlay network from the effects of the BATMAN-adv mesh network presented in Experiment 2 (see Section 4.1.1.2 on page 87). Figure 4.10 presents the two scenarios: on the left are two Raspberry Pi nodes connected using a wireless access point. Processes in this configuration run directly on the nodes and communicate using this network. On the right, an overlay network has been constructed on top of the existing wireless network using the Docker Swarm container orchestration tool. Processes in this configuration are run inside containers and communicate over the virtual network. The dependent variable in this experiment is the QoS metric measurements, while the independent variable is the communication network exposed to the process: physical vs overlay. The same software suite used in the previous two experiments is used to measure the QoS metrics.

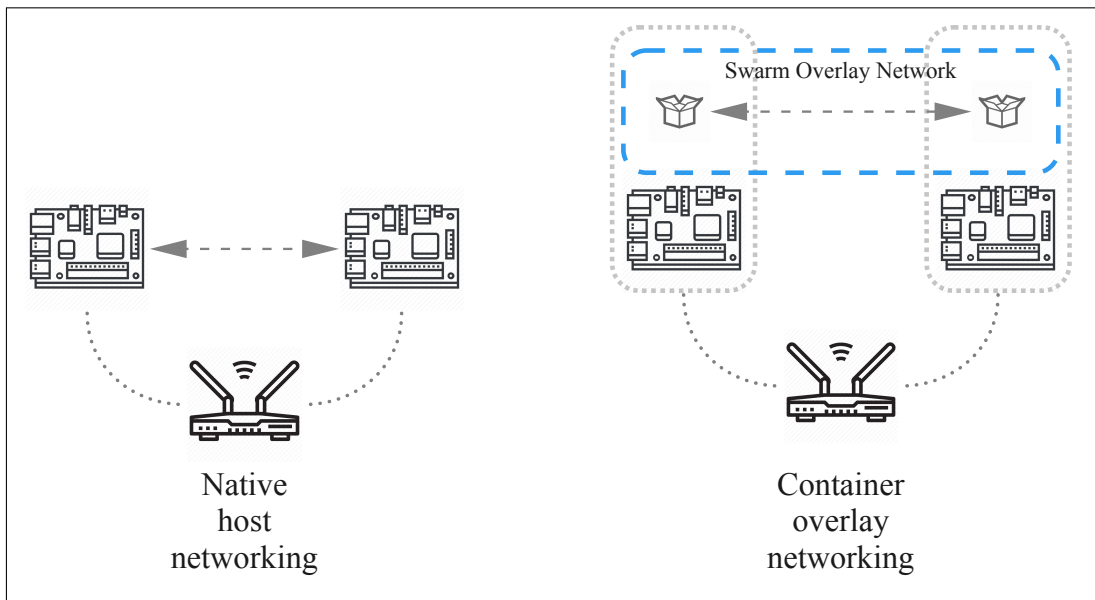


Figure 4.10 Comparison of native host networks versus Docker Swarm overlay network.

Interface
wlan0 (IEEE 802.11), phy 0, reg: US (DFS-FCC), SSID: raspi-mesh
Levels
link quality: 100% (70/70)
=====
signal level: -17 dBm (0.02 mW)
=====
Statistics
RX: n/a
TX: n/a
Info
mode: IBSS, joined IBSS: 1E:DF:73:4B:C7:A4
freq: 2412 MHz, channel: 1 (width: 20 MHz (no HT))
station flags: (none), preamble: short, slot: long
power mgt: off, tx-power: 30 dBm (1000.00 mW)
retry: short limit 7, rts/cts: off, frag: off
encryption: off (no key set)
Network
wlan0 (UP BROADCAST MULTICAST)
mac: 00:0F:00:39:A8:6D, qlen: 1000
ip: n/a, mtu: 1528

Figure 4.11 Example output of the *wavemon* utility used for displaying live wireless networking information when connected in a mesh. In this example, it is presenting the information on the *raspi-mesh* SSID, but it was also used to measure signal quality in tests using infrastructure wireless modes.

The experiment was conducted using three Raspberry Pi's connected to a Tenda AC6⁴ access point. This dual-band access point supports 802.11ac in the 5 GHz frequency, but was configured to only broadcast on its 2.4 GHz (IEEE 802.11n) frequency as the mesh-enabled adapters used do not support 802.11ac. The three nodes were connected to the AP and positioned ~1m from it. The signal level was measured using the *wavemon* utility⁵ and all three nodes reported a link quality of 100% (see Figure 4.11).

Figures 4.12 to 4.15 on pages 93–95 present the measurements of QoS metrics during point to point transmissions. In each of these experiments, tests were performed between all three nodes, for example, the first test would be between node one and node two, then node one and node three and finally node two and node three. The results of each individual test were tabulated and processed to produce the results of the experiment. For all of the tests, the environment was kept identical: the only change made was switching between the standard IEEE 802.11n wireless network in infrastructure mode and a virtual overlay network constructed on top of it. The intent was to measure the impact of this virtual network on QoS metrics.

The first test measured bandwidth using *iperf3*. One node would host a server using *iperf* in server mode while the other would connect to that server using *iperf* in client mode. The second node would use the TCP protocol to transfer as much data as it could in 30 seconds. The result of each test series was tabulated and plotted to produce Figure 4.12 on the following page.

⁴<https://tendacn.com/en/product/ac6.html>

⁵<https://github.com/uoaerg/wavemon>

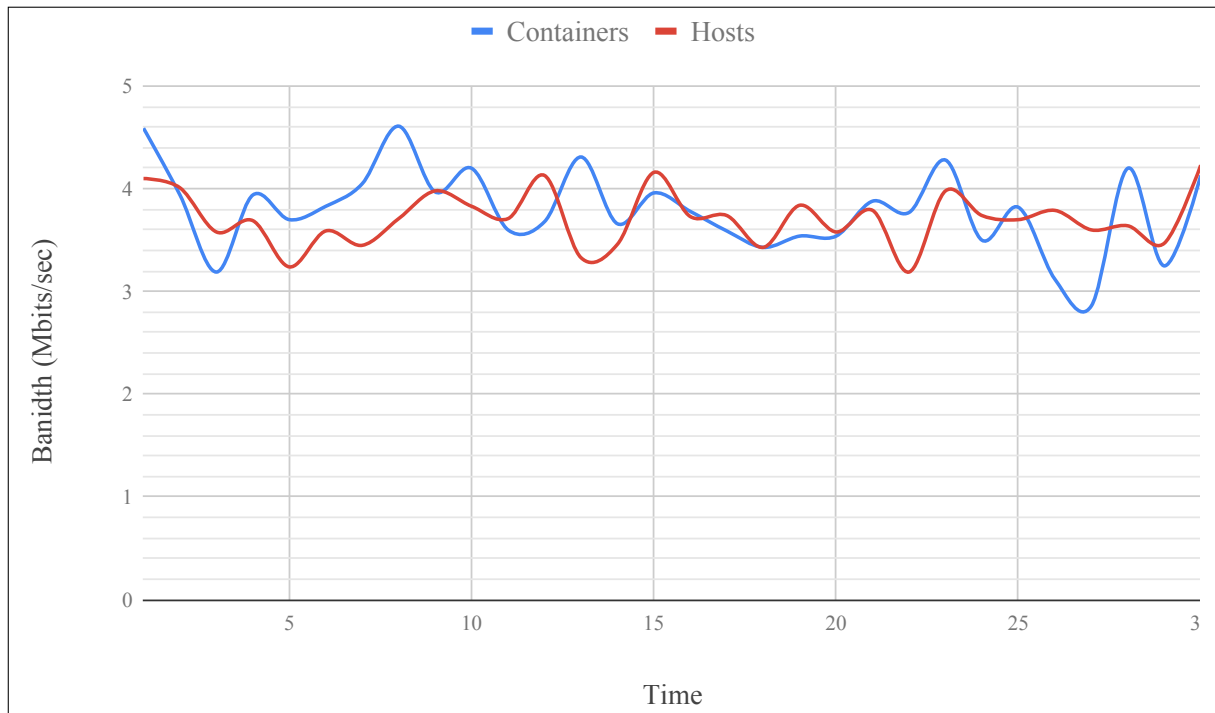


Figure 4.12 Bandwidth comparison between direct communication over 802.11n network and the Docker Swarm overlay network.



Figure 4.13 Latency comparison between direct networking over 802.11n network and Docker Swarm overlay network.

The red line indicates the average bandwidth measurement for host machines and achieves an average bandwidth of 3.71 Mbits/sec. The blue line represents the average bandwidth achieved using containers on separate hosts performing the same test while communicating over an overlay network. The two containers achieved an average of 3.8 Mbits/sec. The result suggests that at this network performance level, the impact of the Docker overlay network does not have a measurable impact on bandwidth.

The second test measured latency using *fping*. For this test, one node sends the standard 56 byte ICMP packet to the designated IP, once every second for 30 seconds. The *fping* utility reports the latency of each transmission. The test was repeated for each pair of nodes and containers, and the results were tabulated and plotted to produce Figure 4.13 on the previous page. As in the bandwidth experiment, the red line represents the average latency of hosts, while the blue line represents the average latency of containers.

The average latency measured between hosts is 3.65 ms compared to the 3.06 ms measured between containers. The result presented by the containers is also slightly more consistent with a lower σ , 0.11 ms vs. 0.16 ms for the hosts. As with the bandwidth test, the results indicate that the influence of the overlay network at this performance level will likely be unnoticeable.

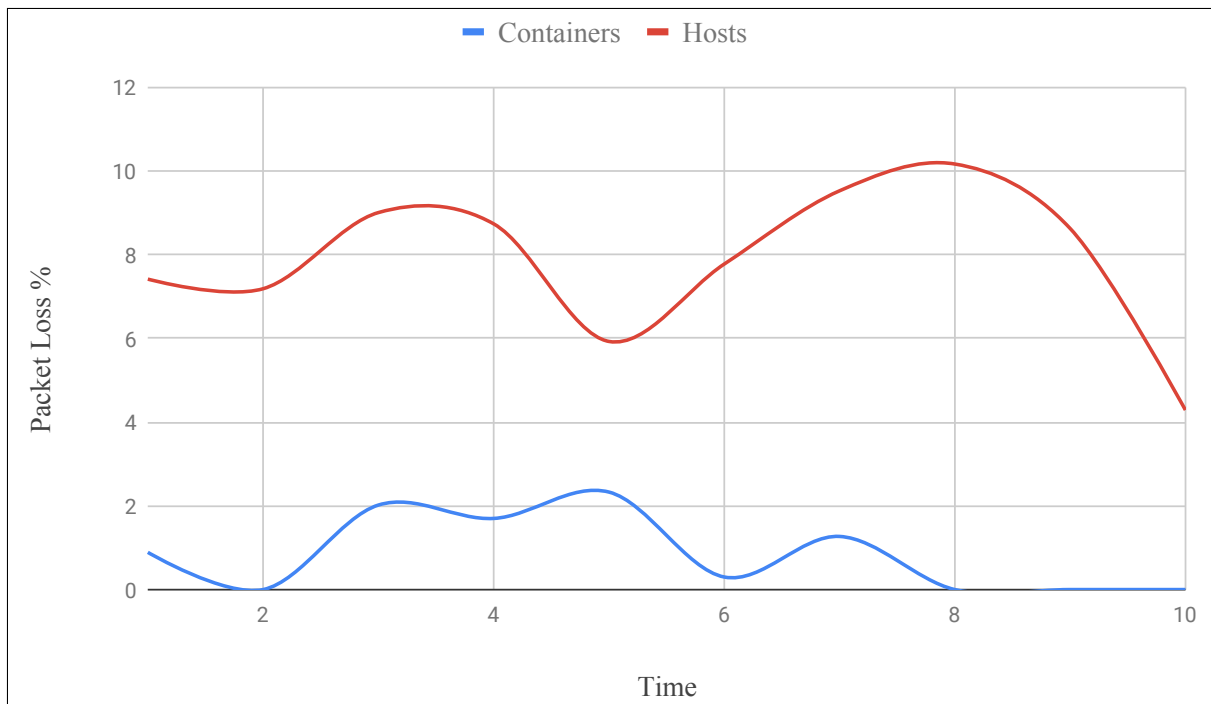


Figure 4.14 Packet loss % comparison between direct networking over 802.11n network and Docker Swarm overlay network.

The third test measured both packet loss ratio and jitter using *iperf3*. For this test, one node would host a server using *iperf3* in server mode while the other would initiate a UDP transmission with *iperf3* in client mode. This command would transmit at 5 Mbits/sec using UDP to the other node

and report what portion of the datagrams was lost as a percentage along with the variance in latency (jitter) in milliseconds. The test was repeated for all three node pairs using both the native network and the overlay network. The results were tabulated and plotted to produce Figures 4.14 on the previous page and 4.15.

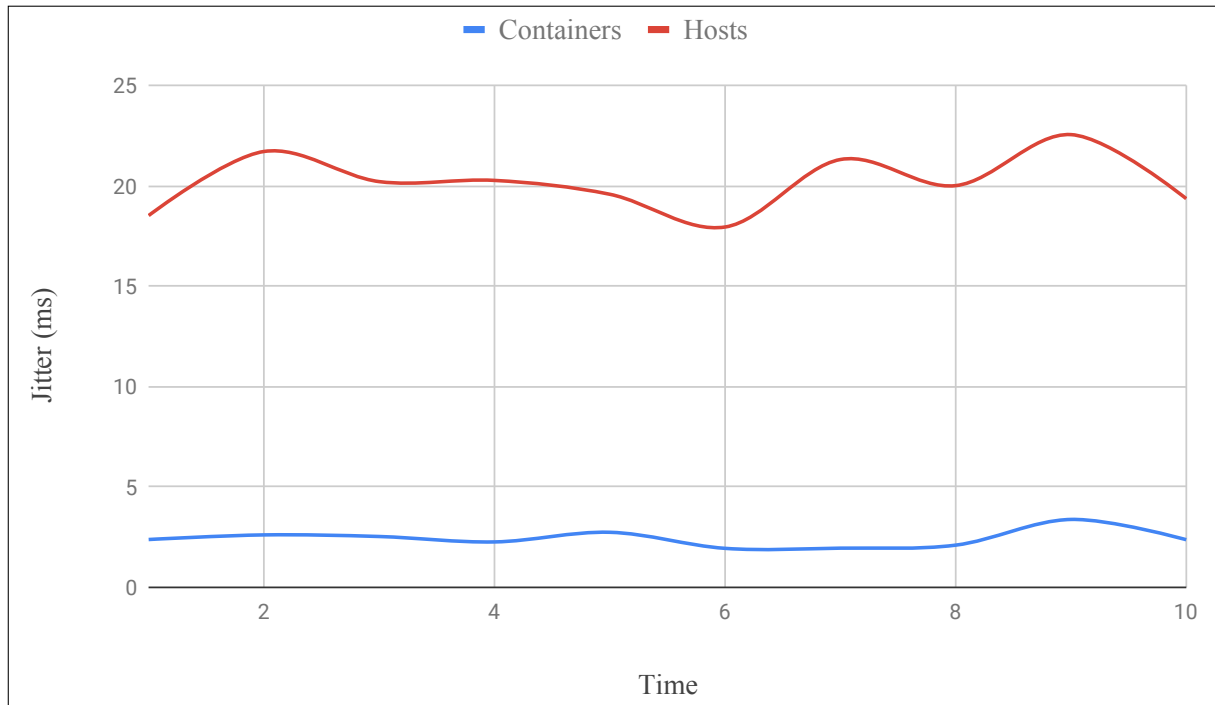


Figure 4.15 Jitter comparison between direct networking over 802.11n network and Docker Swarm overlay network.

These plots reveal a significantly better result for the overlay network over the standard network, both in terms of the packet loss ratio and jitter. This result was very unexpected as the assumption was that the overlay network built on the physical network would carry some overhead and perform worse. The result was the opposite: containers communicating via the overlay network performed better. A possible explanation for this unexpected result is in the functioning of the Docker overlay network. Even though the *iperf3* software running within the containers sent UDP datagrams without setting up the connection first, the Docker overlay uses a VXLAN interface that maintains routes between hosts. The swarm orchestrator is designed for application deployments that might span multiple data centres, and the routing complexity is abstracted away from applications running in the containers. Despite *iperf3* sending UDP datagrams, they are encapsulated in VXLAN packets which enforces different transport characteristics. While the comparison between hosts and containers presents this unexpected result for UDP transmissions, the comparison is valid when considering the result from an application perspective.

This anomaly of nearly 0% packet loss when using the overlay network resembles the 0% packet loss measured when comparing the mesh and infrastructure networks in Section 4.1.1.2. Both

batman-adv and Docker’s overlay network encapsulate standard network traffic and as such, the QoS metric measurements do not reflect what is expected from UDP transmissions. Further investigation into how to more accurately measure and represent network performance in environments that encapsulate standard protocol packets presents an opportunity for future work.

4.1.2 Event-streaming experiments

In this section, the evaluation of the NATS Streaming architecture will be presented. As the communication base for the wireless distributed computing platform, it is important to determine its performance. NATS provides an open-source benchmarking tool called *nats-bench*⁶. This tool was used to measure message throughput and network throughput as the dependent variables. Independent variables include the message sizes, the number of publishers/subscribers on the node running the test and the event-persistence mode. The experiment was conducted in the mesh networking setup as presented in Figure 4.5 on page 88.

A script was developed to automate the running of individual tests. This script was containerised and deployed on the platform with its dependencies. The script runs the *nats-bench* executable for all permutations of the variables, parses the results, performs statistical processing and generates CSV files for analysis and plotting. The results are stored in a Docker volume mounted on the host to make it simple to access the results from the host file system over SFTP, rather than having to extract it from the running container.

Script 4.1 presents an extract of this script’s code, with the full script available from the project repository⁷. The *shelljs* package for NodeJS was used to run shell commands from JavaScript, as shown on line 2. Lines 3 to 7 present the varied parameters provided to the *nats-bench* executable. This command was run in a loop to produce all the permutations of the experiment’s variables and aggregate the results into a large data set for analysis.

```

1  const { stdout, stderr, code } =
2    shell.exec('./bench
3      -s ${t.Server}
4      -ns ${t.Subscribers}
5      -np ${t.Publishers}
6      -n ${t.Messages}
7      -ms ${t.MessageSize}
8      -csv ${filename} ${topic}',
9    { silent: true }
10 );
```

Script 4.1 Extract from *nats-bench* automation script. The command executes the *nats-bench* utility with parameters, ready to process and aggregate the results.

⁶https://nats-io.github.io/docs/nats_tools/natsbench.html

⁷<https://github.com/prossouw79/research-artefact/blob/master/nats-bench/index.js>

An example execution of this script would run three repetitions for each of the three container hosts (line 3), and message sizes (line 7) were varied from 2^0 to 2^{12} bytes. The number of messages (line 6) is calculated to keep the total data transmission constant between different message sizes, e.g. if 100 messages were sent of size 1024 bytes, 200 would be sent of size 512 bytes. The number of publishers (line 5) is varied between 1, 2 and 3 while the number of subscribers (line 4) is varied between 0, 1 and 2. The results of each individual test are captured in a CSV file (line 8). The topic was kept constant across all permutations of the benchmark parameters. Once all the tests in the experiment were completed, the results are aggregated to produce a large data set. This data set is then used to investigate the relationships between dependent and independent variables.

4.1.2.1 Experiment 4: Message size

The first relationship investigated is that of the message size versus the message and network throughput. For this experiment, the data set was filtered to include tests that have one publisher and no subscribers. The reason for this choice is so that the *nats-bench* client only sends messages to the NATS Streaming server and does not need to receive messages. The mean and the standard error values are calculated for both the message throughput and network throughput at each message size. The results are plotted and presented in Figure 4.16. Blue bars represent network throughput in bytes/sec and the red line represents message throughput in messages/sec. The maximum transmission unit (MTU) is indicated on the graph at 1536 bytes - the maximum supported by the adapters. Both plots include bars to represent the standard error of the mean.

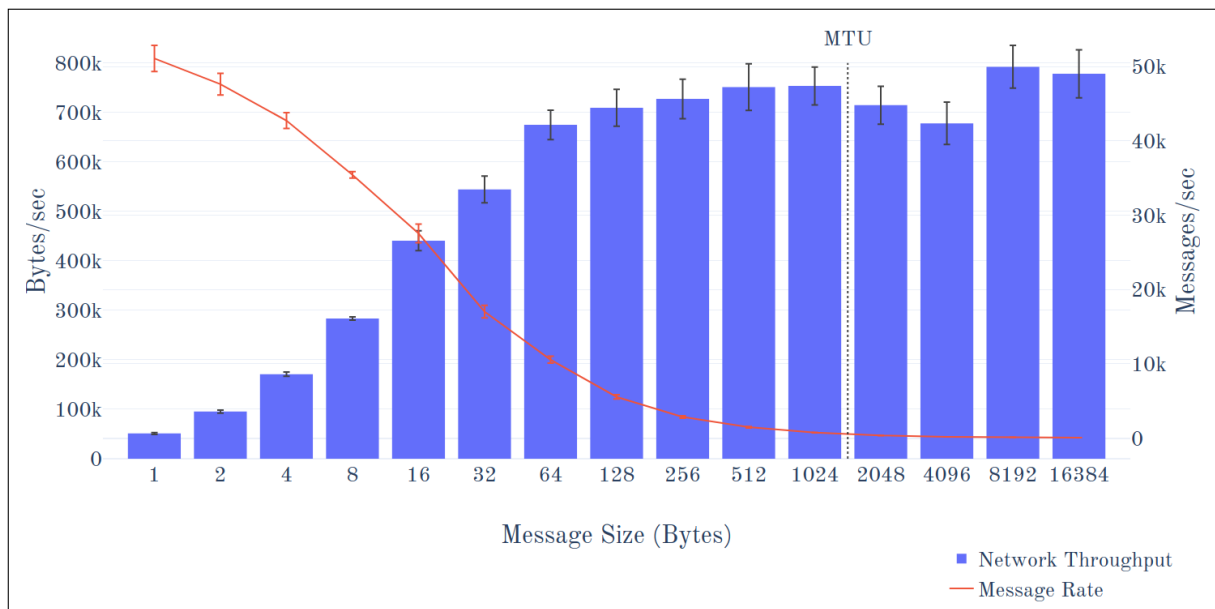


Figure 4.16 Relationship between message size and throughput. The y-axis on the left presents the network utilisation with blue bars indicating the values. The y-axis on the right presents the message throughput and is indicated by the red line.

There is an inverse relationship between the network throughput and the message rate, as expected,

given the limited channel capacity. There is a noticeable reduction in network throughput beyond the MTU value of 1536 bytes at 2048 and 4096 bytes, but a return to the trend increase at 8192 and 16384 bytes. Beyond 1024 bytes, the message and network throughput reaches a plateau. The increase in message size from 1024 bytes to 8192 bytes presents network throughput results that are within the error margin of each other and the plateau correlates with the mesh network bandwidth results presented in Figure 4.6 on page 89. For the remainder of experiments, results were used with a message size of 1024 bytes as there was no significant increase in network throughput beyond this point.

4.1.2.2 Experiment 5: Number of publishers and subscribers

The second relationship investigated is between the number of publishers/subscribers and the message throughput. For this experiment, the data set was filtered to include only tests with a message size of 1024 bytes and then grouped by the combination of publishers and consumers. Figure 4.17 presents these results. On the X-axis, results are grouped by the number of publishers: 1, 2 and 3. Each grouping has a set of results for 0, 1, 2 and 3 subscribers. Error bars represent the standard error values of each result. On the Y-axis, the publish rate is shown with the highest value of 738 messages per second reached with 1 Publisher and no Subscribers. The publish rate of this result is high because the *nats-bench* client only transmits messages, but does not need to receive messages back. In all the other tests, the available network throughput is divided between multiple publishers and/or subscribers.

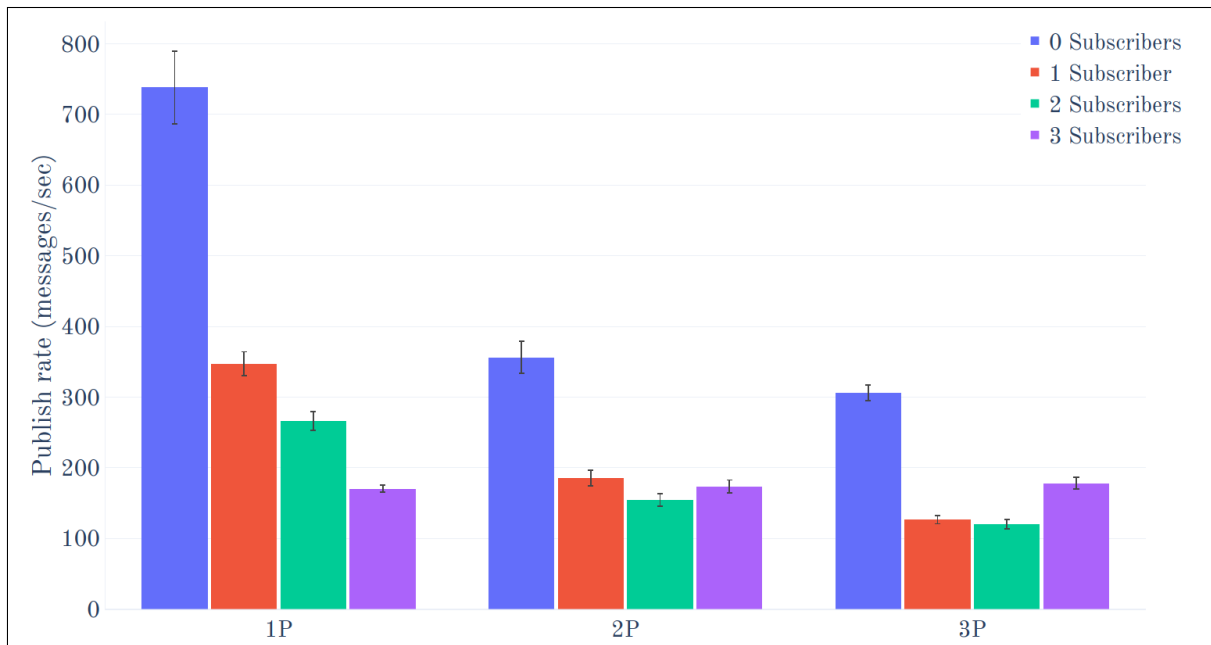


Figure 4.17 Comparison of message rates between pairs of publishers and subscribers linked to the same topic on the same node. This test was run at a fixed message size of 1024 bytes and results are grouped by the number of publishers (1P, 2P and 3P).

From the results, it is evident that the number of publishers and subscribers has a significant effect on message throughput, as the available network throughput is shared between the clients. This is the expected outcome as the 802.11 wireless network transmissions are currently half-duplex: communication is enabled for both sending and receiving packets, but not simultaneously [8]. The IEEE 802.11 working group started planning Full-Duplex communication (simultaneously transmit and receive signals on the same frequency) in May of 2018 [142], but the functionality has not reached consumer devices.

4.1.2.3 Experiment 6: Event persistence

In this section, the investigation into the impact of the persistent storage used by the NATS Streaming server is presented. NATS Streaming will persist messages in memory by default. This approach provides limited persistence of events; if a client temporarily disconnects, it can replay the event stream and get all events, but if the server process is stopped, all events are lost. An alternative is to use non-volatile file storage for events. The files contain all the state that the NATS Streaming Server requires to restore itself if the process is restarted. The volatile memory of a device is much faster than its non-volatile storage and this experiment will investigate whether a persistent event store has a measurable impact on message throughput.

Configuration for MEMORY persistence:	Configuration for FILE persistence:
<pre> 1 listen : 0.0.0.0:4222 2 write_deadline: 10s 3 # store: file 4 # dir: datastore 5 # file_auto_sync: 1 6 # file_sync: true 7 debug: true </pre>	<pre> 1 listen : 0.0.0.0:4222 2 write_deadline: 10s 3 store: file 4 dir: datastore 5 file_auto_sync: 1 6 file_sync: true 7 debug: true </pre>
NATS start with MEMORY persistence:	NATS start with FILE persistence:
<pre> 1 Recovering the state... 2 No recovered state 3 Message store is MEMORY 4 ----- Store Limits ----- 5 Channels: 100 * 6 ----- Channels Limits ----- 7 Subscriptions: 1000 * 8 Messages : 1000000 * 9 bytes : 976.56 MB * 10 Age : unlimited * 11 Inactivity : unlimited * 12 ----- 13 Streaming Server is ready </pre>	<pre> 1 Recovering the state... 2 Recovered 1 channel(s) 3 Message store is FILE 4 Store location: datastore 5 ----- Store Limits ----- 6 Channels: 100 * 7 ----- Channels Limits ----- 8 Subscriptions: 1000 * 9 Messages : 1000000 * 10 bytes : 976.56 MB * 11 Age : unlimited * 12 Inactivity : unlimited * 13 ----- 14 Streaming Server is ready </pre>

Figure 4.18 Corresponding configuration files and NATS Streaming startup output.

Figure 4.18 on the previous page presents the experiment setup: two configuration files were prepared to isolate the persistence mode. The files are identical, except that in the case of the MEMORY persistence configuration file (top-left), the options for file event storage (lines 3-6) are commented out. These lines instruct the NATS Streaming server to store events in a directory called *datastore* and synchronise events to disk every second. Line 7 instructs the server to print debugging information to console. An important option set in both configuration files is the write deadline option. The default write deadline for events is two seconds, but with event sizes beyond the MTU value of 1536, individual tests start to fail. Reviewing the NATS Streaming Server container logs revealed the cause:

```
[INF] ::1:63283 - cid:7 - Slow Consumer Detected.
```

When large messages are published with multiple publishers and subscribers needing to send and receive the message over the same connection, the connection is terminated by the NATS Streaming server. The termination is intended behaviour as the NATS Streaming server blocks slow subscribers. These clients are disconnected to prevent a build-up of *back-pressure*. The rationale behind this function is to protect the server and the integrity of the messaging system [143]. In effect, only subscribers that are able to “*keep up*” are allowed to stay connected so that the real-time aspect of event-streaming is prioritised. In order to allow tests with large message sizes and multiple publishers and subscribers on the same client, the write deadline was increased to 10 seconds.

The comparison between memory and file-based message persistence involved running the same test suite used in the previous experiment and changing the configuration file provided to NATS Streaming. In this use of NATS, the low network throughput available in the mesh network is the bottleneck as both processor and memory utilisation of the container was low. The mesh network presented an average throughput of ~9 Mbits/sec when tested with *iperf3*, presented in Table 4.1 on page 88. This should allow for ~1125 bytes/sec and the highest average throughput measured was 835 bytes/sec, 74% of the *iperf3* result.

As presented in Figure 4.18 on the previous page, the startup output reports a recovered state from the *datastore* directory (bottom-right, line 2). Figure 4.19 on the following page presents the results of this experiment. The blue bars represent the average publisher message rate, whereas the red bar represents the average subscriber rate. The results are grouped based on their event-persistence mode with memory persistence on the left and file persistence on the right. This experiment did not reveal any measurable impact of the event-persistence mode on message throughput in this use-case. In fact, the file-persistence results presented a slightly higher message throughput with a higher standard error value, but both results were within the margin of error. A theory for this result is that the messages sent/received over the mesh network do not represent a high enough data rate that the storage speed of the Raspberry Pi nodes has a noticeable effect. If the network throughput approaches the storage speed, it could possibly become a bottleneck, but this is unlikely when using a wireless network.

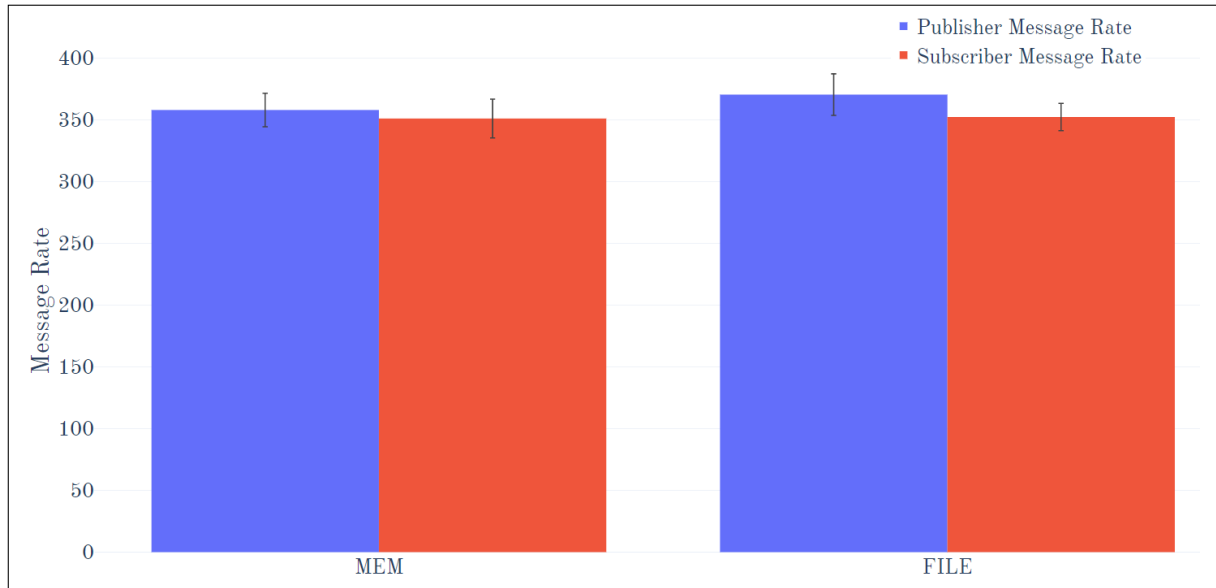


Figure 4.19 Comparison of message rates between memory-persistence and file-persistence modes of the NATS Streaming Server.

4.2 Summary

In this chapter, the work of Peffers et al. [38] was used to select an evaluation method appropriate for the research artefact. The selected method was technical experiments to complement the illustrative scenario method used to evaluate iterations in Chapter 3. The wireless networking and event-streaming aspects of the artefact were evaluated using experiments to measure its performance and model the relationships between selected dependent and independent variables.

Experiment 1 on page 84 investigated the effect of transmission distance on the four QoS metrics. Experiment 2 on page 87 compared the QoS metrics of typical infrastructure-mode wireless networking to the platform’s mesh networking selected for its increased fault tolerance. Experiment 3 on page 91 investigated the impact of Docker Swarm’s overlay networking on network performance.

Experiment 4 on page 97 modelled the relationship between message size and the message throughput/network utilisation of the event-streaming implementation used by the artefact. The effects of varying numbers of publishers and subscribers connected to a NATS Streaming server were investigated in Experiment 5 on page 98. Finally, the impact of two event-persistence mechanisms provided by NATS Streaming on message throughput and network utilisation was investigated in Experiment 6 on page 99. The key findings of these experiments are summarised in Section 5.2.

Chapter 5

Conclusion

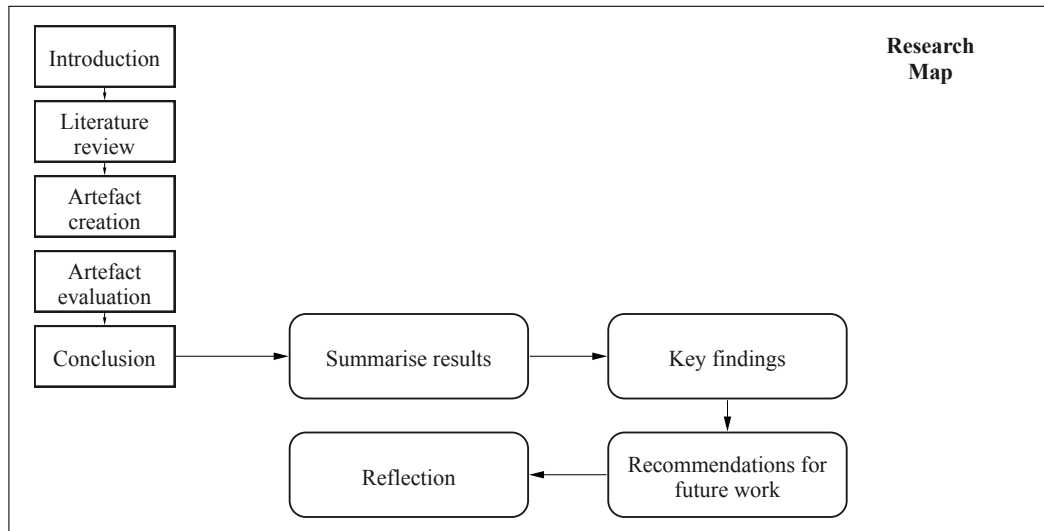


Figure 5.1 Research map extract for Chapter 5. The complete research map is presented on page 1.

This research was conducted to answer the research question presented in Section 1.3: *“How can existing system architectures be used to create a fault-tolerant wireless distributed computing platform?”* The central theoretical fields of wireless networking, distributed computing and system architecture were investigated to support the creation of the platform presented in Figure 1.3 on page 9. In Chapters 2, 3 and 4, the theoretical and practical efforts made to each of the aims and objectives set in Section 1.4, are documented. These efforts were guided by the research approach detailed in Section 1.6.

The DSR methodology includes the use of the engineering cycle, presented in Figure 1.4 on page 11, which guided four iterations in a systems architecture process. The work of Peffers et al. [38] was used in Section 1.6.3 to determine which methods are best suited to evaluate the artefact, leading to the conclusion that both technical experiments and illustrative scenarios are appropriate methods. Each iteration includes a combination of design problems and knowledge questions which had to be overcome. The design problems were solved with the application of system architecture while knowledge questions were addressed by reviewing relevant literature.

Completing the cycles resulted in a platform optimised for fault tolerance. The iterative process followed during the creation of the artefact is documented in Chapter 3. The illustrative scenario evaluation method was applied during the development of each iteration to evaluate the platform’s fault tolerance, while supporting a simulated drone swarm application. Completing this process produced the desired DSR outputs of this research:

- an instantiation: *“The structure and organisation of a system’s hardware or software or*

part thereof”. This structure is presented in Figure 3.21. The instantiation serves as a functional platform to investigate fault-tolerant wireless distributed computing for a robot swarm application; and

- a method: “*Actionable instructions that are conceptual (not algorithmic)*”. The method is the combination of efforts to improve fault tolerance at various levels, summarised in Table 3.8. It combines IEEE 802.11s mesh networking with the BATMAN-adv routing protocol, Docker Swarm container orchestration and NATS event streaming into a single platform capable of supporting fault-tolerant distributed computing.

Table 3.8 on page 81 summarises the culmination of design choices made during the system architecture process. It details the various layers of the platform with descriptions of each layer’s contribution to the platform’s fault tolerance. The application of the second evaluation method is presented in Chapter 4. Technical experiments were performed to evaluate the performance of the wireless mesh network and event-streaming aspects of the platform. A summary of the research results is presented next.

5.1 Result summary

The primary DSR output of this research is the instantiation of the fault-tolerant wireless distributed computing platform. An iterative development process that aimed to improve fault tolerance in subsequent prototypes was conducted, as documented in Chapter 3. The scenarios in which the platform would demonstrate its fault tolerance are listed in Section 3.6.

This artefact’s performance was evaluated using technical experiments, which are documented in Sections 4.1.1.1 to 4.1.2.3. In experiment 1, the impact of transmission distance on network performance was measured. The experiment found that network performance suffered in cases where the antennas were in close proximity to each other. Network performance also declined at a distance beyond 250 cm. All further experiments were conducted at a distance of ~150 cm between nodes in an attempt to minimise the impact of transmission distance on the results. At this distance, the signal quality reported by the clients was 100% as shown in Figure 4.11 on page 92.

A design choice was made in Section 3.3 that favoured the use of IEEE 802.11s wireless mesh-networking over traditional IEEE 802.11n infrastructure networking. This was done in order to eliminate the single point of failure that the use of an access-point device would introduce. The experiments conducted in Section 4.1.1 show that this choice not only enhanced the fault tolerance of the platform, but provided higher network performance.

The results summarised in Table 4.1 on page 88 show that on all four of the Quality of Service (QoS) metrics, the wireless mesh networking solution outperformed the infrastructure networking solution. The packet loss measurements conducted in Experiment 2 consistently presented an unexpected result of exactly 0% which is not in keeping with literature. A similar anomaly was apparent in Experiment 3 which aimed to measure the impact of Docker’s overlay network driver on

network performance. The result, presented in Figure 4.14 on page 94 once again showed packet loss measurements that were low enough to draw suspicion. Plausible theories that attempt to explain this phenomena are presented in Sections 4.1.1.2 and 4.1.1.3. A common characteristic with the mesh-networking experiment is that in both cases, the UDP packets used to measure packet loss were encapsulated. In the case of the mesh network, BATMAN-adv implemented packet routing at the Data Link layer and encapsulated packets in Ethernet frames. In the case of the Docker Swarm overlay network, packets were encapsulated in VXLAN packets. In both cases, a theory to explain the measurements is provided, but this phenomenon warrants future research into methods to conduct network performance measurement in environments where packets are encapsulated. Standard tools, such as *iperf3*, do not seem to be suitable in such environments.

The performance measurements of the event-streaming aspects of the platform are presented in Section 4.1.2. Experiment 4 measured the correlation between message size and the message and network throughput. The result is as expected, given a limited network throughput: sending larger messages reduces the message rate. It is only at ~256 bytes that the message rate starts to be limited by network throughput and message sizes beyond the MTU values set by the adapter seem to reduce the network throughput. This phenomenon is visible for message sizes of 2048 and 4096 bytes, but does not seem to have a measurable effect at 8192 bytes.

Experiment 5 measured the impact of running multiple producers and subscribers on message throughput. All of the stream processors running on any given node need to share their available network throughput and therefore running multiple instances of producers and consumers limits the message rate of each individual stream processor.

Experiment 6 attempted to measure the impact of two persistence modes available in NATS Streaming. Events can be persisted in memory or in a file structure and since storage is substantially slower than memory, the impact of file-based event persistence was investigated. It was found that the storage speed of the nodes did not have a measurable impact on the message throughput in comparison with memory. An explanation for this is that the network throughput does not approach the storage throughput.

Given the results of the experiments and the aims set for this research, the platform demonstrates meaningful progress towards improving fault-tolerance. The final iteration of the platform supports wireless distributed computing, presents fault-tolerant behaviour in multiple scenarios and can support a robot swarm application.

5.2 Key findings

In this section, the key findings of the research are summarised in two parts. The first part, presented in Section 5.2.1, summarises noteworthy findings produced during the practical development phase of the artefact, as well as findings relating to the enhancement of fault tolerance. The second part, presented in 5.2.2, summarises the findings encountered while evaluating the research artefact using

technical experiments and is focused on the performance of the artefact.

5.2.1 Fault-tolerance enhancements of the WDC platform

In this section, the insights gained in creating a fault-tolerant wireless distributed computing platform for a robot swarm application are summarised. The artefact's architecture is represented in layers which are used to provide a structure for this section.

At the hardware level, three low-cost and low-power Raspberry Pi nodes are used to support the rest of the platform. At this layer, fault tolerance is enhanced with redundancy: all the software components are able to run on one, two or all three nodes. Using a single node introduces a single point of failure, so the use of multiple nodes is a basic requirement for enhancing the fault-tolerance of a platform.

At the networking level, IEEE 802.11s wireless mesh networking adds redundant routing paths between nodes while eliminating the single point of failure that the use of a central access point introduces. This also allows a drone swarm to reach a physical area larger than a single access point can cover.

At the software level, four iterations of progressively more fault-tolerant wireless distributed computing platforms were developed. The first iteration used MPICH and established a baseline wireless distributed computing platform. While the MPI paradigm is very well established within the distributed computing field, it did not demonstrate fault tolerance in an environment where network connectivity between nodes cannot be guaranteed.

In the second iteration, an attempt was made to improve upon this by replicating the entire application state across all available nodes using WebSockets. When a node was disconnected, only the application instance running on that node suffered. This was an improvement in fault tolerance over the MPICH implementation, but the distributed application more closely resembled a distributed information system than a distributed computing system. This shortcoming was eliminated in the third iteration by splitting the application into client and server components. Each client only stored a local copy of the distributed application state and relied on one of multiple redundant RPC compute server instances to perform most of the required processing. The clients were designed to recover from the temporary unavailability of compute servers. The use of asynchronous execution greatly enhanced fault tolerance, as the threads in the client application were not blocked when a compute server instance was unavailable. This restricted the system fault to only that specific call and subsequent calls to other servers would complete successfully.

In the final software iteration, the decoupling of state storage and computation was extended by redesigning the application to use an event-streaming architecture. State storage and event transport are left to purpose-built message brokers: Apache Kafka and NATS Streaming. The processing is deployed as a set of decoupled stream processors that subscribe to a topic, perform a function on the events in that topic and publish the result onto a different topic. This approach proves to be

the most fault-tolerant and NATS Streaming was selected over Apache Kafka because of Kafka's significant memory usage.

At the deployment level, the use of containerisation and container orchestration was investigated. Container orchestration greatly improves fault tolerance by deploying the application as a set of replicated services that can be dynamically provisioned across the available container hosts. Docker Swarm is used as the container orchestrator and greatly simplifies service replication and configuration. This enhances fault tolerance as configuration errors are a major source of system faults. If a process in a container is terminated due to an error, Docker automatically detects it and re-provisions a new container with the same image.

In summary, the key design decisions in the system architecture process that contributed to the fault tolerance of the wireless distributed computing platform include:

1. Redundant nodes that perform computation;
2. Redundant network paths between nodes;
3. Replication of services that perform computation;
4. Asynchronous execution of computation;
5. Decoupling of the distributed application state and distributed computation;
6. Decoupling of the state storage/transport and state transformation;
7. Use of a micro-service architecture that promotes the decoupling of an application into single-responsibility components that can be easily replicated.
8. Use of an event-streaming architecture that decouples state storage/transport from the processing.
9. Use of container orchestration as an application deployment method to facilitate service replication, abstracted networking and fault detection.

These choices produced a platform that exhibits fault-tolerant behaviour. The secondary evaluation method investigated the performance that the artefact provides as a platform for wireless distributed computing.

5.2.2 Performance evaluation of the WDC platform

In Section 4.1.1, the technical experiment method of evaluation was applied to the research artefact to determine the QoS metrics of the wireless mesh network and the message throughput of the event-streaming platform. Six experiments were performed, which produced key findings with regard to the performance of the platform. The findings from the six experiments are:

1. Transmission distance has a profound effect on IEEE 802.11 wireless network performance. When the transmitter and receiver are too close together (<50 cm), there is a clear reduction in network performance. Distances beyond 300 cm also introduce a reduction in average network performance and consistency. This result is presented in Figure 4.3 on page 86.
2. There is a substantial difference in the network performance of infrastructure and mesh

networks. The mesh network delivers significantly better results in all four QoS metrics, as presented in Table 4.1. However, this result could likely be limited to the combination of hardware used in this research and does not imply that mesh networks always perform better than infrastructure networks. The packet loss measured when using the mesh network was consistently 0%, a result that required scrutiny. Two theories for this unexpected result were presented, but this phenomenon is a good candidate for further investigation.

3. The introduction of containerised processes and overlay networks were investigated. At the performance level of the infrastructure wireless network, the overlay network introduces no measurable reduction in network performance. In fact, the packet loss measurements of container networking present significantly better results than the native networking as presented in Figure 4.14 on page 94. Similar to the previous point, a potential explanation for this result is provided, but it requires further investigation.
4. The impact of message size on event-streaming message throughput was investigated and is presented in Figure 4.16 on page 97. As expected, the message throughput decreases as the message size increases, given a bounded maximum network throughput. The network utilisation shows a local peak just below the MTU value of 1536 bytes. Message sizes beyond the MTU have to be split up in multiple 802.11 frames and visibly reduce the network utilisation for 2048 and 4096 bytes. It is interesting to observe the network utilisation return to the trend for message sizes of 8192 and 16384 bytes. An explanation for this phenomenon is not provided in this research and is an opportunity for further investigation.
5. The number of publishers and subscribers has a significant effect on the message throughput over wireless links as presented in Figure 4.17 on page 98. Current wireless networking equipment is limited to half-duplex transmissions and therefore switching between transmission and reception has an impact when there is both a publisher and subscriber on a given node.
6. The impact of event persistence in NATS Streaming was investigated and the default in-memory storage was compared to file-based storage. Figure 4.19 on page 101 shows that file-based persistence introduces no measurable performance impact. A theory for this result is that the network throughput does not come close to saturating the storage throughput of the hardware (SD Card), and therefore file storage can easily keep up with the rate of events to be stored.

The platform developed in Chapter 3 is considered a success when considering the aims and objectives set in Section 1.4. The problem statement presented in Section 1.3 was *“How can existing system architectures be used to create a fault-tolerant wireless distributed computing platform?”*. In this section, the question was successfully answered. However, there are important aspects within the broader area of research that provide opportunities for expansion. These opportunities are presented in Section 5.3.

5.3 Future work

The findings of this research produces some insight into the underexplored research area of fault-tolerant wireless distributed computing for robot swarms. However, there are opportunities for expansion. In this section, the opportunities that are deemed worthwhile, but could not be investigated within the scope of this research are presented.

5.3.1 Investigate the QoS metrics of multi-hop transmissions

One of the key benefits of using mesh networks is the redundant routing paths that are available. If two nodes are out of range for direct transmissions, they might still be able to communicate by relaying the packets through multiple hops of nodes that are within range. This functionality was present in the artefact, but not investigated as the nodes were always close enough that a single hop transmission path was available. The experiments conducted indicate that the mesh network delivered better network performance. However, would this be the case if transmissions had to be relayed once, twice or more? An expansion of this research would account for scenarios where nodes cannot transmit over a single hop.

5.3.2 Revisit Apache Kafka with a more powerful SBC

The base hardware layer of the artefact includes the use of three Raspberry Pi Model 3 B+ nodes. The SBC was chosen for its low cost, low power consumption and support. The level of processing power delivered by them was enough to facilitate this research, but is unlikely to be sufficient for the application that inspired it. A notable advantage that Intel's implementation for their drone swarm demonstration (presented in Figure 1.2 on page 4) has over the distributed computing implementation is that the centralisation of processing decouples the concerns of high performance computing and low-powered drones that might favour manoeuvrability and battery life over fault tolerance.

The Raspberry Pi Model 3 B+ was the most powerful model available when the decision of which hardware to used was made. Since then, the Raspberry Pi Model 4 B has been released. This model represents the largest proportional increase in computational power of any generation as shown in Figure 5.2 on the following page. Perhaps most crucially, the latest model is available with 4 GB of memory instead of 1 GB. This increase in memory capacity could make Apache Kafka a viable option for this use-case. While the performance of the platform developed in this research might be a limitation, the performance of small, low-powered and low-cost computers are increasing.

5.3.3 Develop a network performance analysis method for use in environments with packet encapsulation

This opportunity was revealed in the Experiments 2 and 3 when the impact of the *batman-adv* mesh network and Docker swarm overlay network was measured. The packet loss measurements

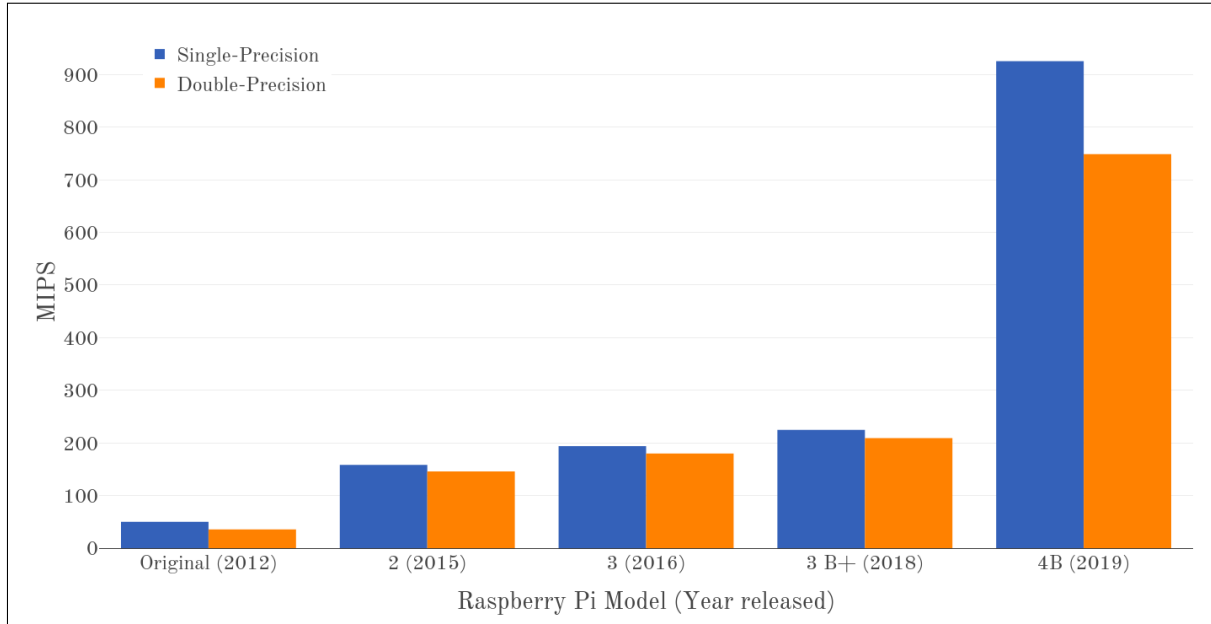


Figure 5.2 Comparison of processor performance in millions of instructions per second (MIPS) between generations of Raspberry Pi models. Figure created using data from original work by Zwetsloot [144].

in particular are “*too good to be true*”. In both these cases, the UDP packets sent by the *iperf3* application are encapsulated. In the case of the mesh network, the *batman-adv* kernel module encapsulates traffic in Ethernet frames before relaying them to the intended mesh node. In the case of Docker Swarm, the overlay network driver encapsulates traffic in VXLAN packets before relaying them to containers that might be running on a separate node. In both cases, the very low packet-loss measurement seems to suggest that the transport protocol used for this relay is more reliable than UDP. It would be worthwhile to investigate a measurement methodology that can present network performance characteristics more accurately.

5.3.4 Compare container orchestrator solutions for low-powered WDC

Shortly after selecting Docker Swarm over the more popular Kubernetes (k8s), k3s¹ was released. This orchestrator is a slimmed-down, yet fully compliant Kubernetes distribution. It would be worthwhile to do a comparative study to establish which orchestrator is better suited to wireless distributed computing. One frustration encountered with Docker Swarm is that the containers that represent services are re-provisioned in the event of a host failing, but when the host comes back, the service does not re-balance containers automatically. This results in an unbalanced cluster with underutilised capacity. The *descheduler* project² for Kubernetes is capable of killing pods (containers in Docker) when a cluster becomes unbalanced. Revisiting the research and comparing different container engines for this use-case would add breadth to this underexplored research area.

¹<https://k3s.io>

²<https://github.com/kubernetes-sigs/descheduler>

5.3.5 Investigate the use of alternative networking standard to improve swarm range

The effects of increased network transmission distance on network performance was investigated in Experiment 1 and it is evident that it has a profound effect. With the specific combination of networking, hardware used in this research networking performance suffered at a relatively small distance of 400 cm. Other combinations of hardware could perform better in this regard, but it could be valuable to investigate the use of alternative networking standards for robot swarm applications. Two standards that have a focus on transmission distance over data rate include *LoRa*³ and *HaLow*⁴. Both standards support mesh networking and might be well-suited to serve as a back-haul connection for drone swarms to operate in a larger area.

5.4 Personal reflection

In this section, a less formal personal reflection on the research is presented. In my opinion, the value of this research is in the developed method: the novel combination of wireless distributed computing, event streaming and wireless mesh network to enhance fault tolerance for a robot swarm application. The enhancements to platform fault-tolerance were fairly comprehensive and included mitigations at the hardware, networking, application and deployment levels. The combination of these efforts resulted in a fault-tolerant platform.

It is my opinion that both the instantiation and method research outputs show potential for further development and optimisation. The application of event-streaming technology to a wireless distributed computing application looks promising. While the simulated flight control application developed in the research might be quite simple, modelling the problem using event-driven architecture seemed much more natural than the WebSocket or RPC implementations.

In this approach, the technological paradigm used to model a digitally constructed reality closely resembled the philosophical basis of this research. A central claim of the positivist paradigm is David Hume's nature of reality: "*reality consists of atomistic and independent events*". This view lies both at the core of empiricism itself and the event-streaming architecture used in this research. It is fascinating to me that the philosophical underpinnings of the research and the artefact produced by the research fit so well.

The available technology within this domain is constantly evolving and access to advanced technology has never been more accessible. The major components used in the artefact have no tight coupling between them and therefore variations of the method can be investigated while retaining the pattern. As an example, during the duration of this research, NATS Streaming went from relative obscurity to a Cloud Native Computing Foundation (CNCF) "*incubating*" project. Another example is the

³<https://lora-alliance.org/>

⁴https://standards.ieee.org/standard/802_11ah-2016.html

initial v1.0 release of CloudEvents⁵, which is an attempt to create a specification for what constitutes an event, as messaging solutions, such as NATS and Kafka, currently lack a common definition.

The CloudEvents project, which aims to develop such a specification, was released mere weeks before the submission date of this research. This enormous pace of innovation is simply the nature of the open-source software field - multiple solutions to common problems competing for developers' mindshare. A cursory overview of the dynamic CNCF landscape⁶ shows multiple alternatives to NATS Streaming and other open-source technologies that could potentially be used to create a fault-tolerant wireless distributed computing platform for robot swarms. Would the platform have been better in some way if it was implemented with DeepStream⁷ rather than NATS Streaming? Possibly, but research needs to be done to answer the question. It is my hope that this research might stimulate further research into using event-streaming architecture in distributed computing and swarm robotics.

The Intel drone swarm demonstration at the opening ceremony of the winter Olympic Games was viewed by millions of people. This demonstration inspired this research and will hopefully inspire further developments within this field. A significant portion of the research was conducted part-time while working as a software developer and the academic research certainly enriched the skills I used daily in a professional capacity. Solving industry problems using novel techniques in my professional capacity also equipped me to approach the research with a strong focus on real-world utility.

This research presented multiple challenges which needed to be overcome; not just as an academic in training but as a professional within the realm of technology. I have gained a fresh respect for research as a craft and hope to continue honing the skills that I have acquired while in the pursuit of knowledge.

⁵<https://github.com/cloudevents/spec>

⁶<https://landscape.cncf.io/>

⁷<https://deepstream.io/>

Bibliography

- [1] “Intel drones light up the sky”. Internet: <https://intel.com/content/www/us/en/technology-innovation/aerial-technology-light-show.html> [Nov. 06, 2018].
- [2] C. Gartenberg, “Intel’s Winter Olympics light show featured a record-breaking 1,218 drones”. Internet: <https://www.theverge.com/2018/2/9/16994638/winter-olympics-2018-intel-drone-show-world-record> [Nov. 06, 2018].
- [3] Oxford University Press, “UAV”, in *Lexico.com*, 2019. Internet: <https://www.lexico.com/en/definition/uav> [Mar. 12, 2018].
- [4] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, and T. Vogel, “Software engineering for self-adaptive systems: A second research roadmap”, in *Software Engineering for Self-Adaptive Systems II*, Springer, 2013, 1–32.
- [5] “Intel breaks drone record with Olympics display with promise of more to come”. Internet: <https://theengineer.co.uk/intel-drone-olympics> [Jun. 22, 2019].
- [6] M. Murison, “Intel’s drones light up the Super Bowl again”. Internet: <https://dronelife.com/2019/02/04/intels-drones-light-up-the-super-bowl-again> [Aug. 06, 2019].
- [7] G. Chmaj and H. Selvaraj, “Distributed processing applications for UAV/drones: a survey”, in *Progress in Systems Engineering*, Springer, 2015, 449–454.
- [8] A. S. Tanenbaum and D. Wetherall, *Computer networks*. Prentice hall, 2011, ISBN: 9780135408001.
- [9] D. Bhattacharyya, T.-h. Kim, and S. Pal, “A comparative study of wireless sensor networks and their routing protocols”, *Sensors*, vol. 10, no. 12, 10506–10523, 2010.
- [10] M. I. Alfaqawi, J. Chebil, M. H. Habaebi, and D. Datla, “Wireless distributed computing for cyclostationary feature detection”, *Digital Communications and Networks*, vol. 2, no. 1, 47–56, 2016, ISSN: 2352-8648.
- [11] D. Datla, X. Chen, T. Tsou, S. Raghunandan, S. S. Hasan, J. H. Reed, C. B. Dietrich, T. Bose, B. Fette, and J.-H. Kim, “Wireless distributed computing: a survey of research challenges”, *IEEE Communications Magazine*, vol. 50, no. 1, 144–152, 2012, ISSN: 0163-6804.
- [12] Oxford University Press, “Robust”, in *Lexico.com*, 2019. Internet: <https://www.lexico.com/en/definition/robust> [Apr. 04, 2016].
- [13] “Robust”, in *Merriam-Webster.com*, 2017. Internet: <https://merriam-webster.com/dictionary/robust> [Jun. 02, 2017].
- [14] K. P. Birman, *Guide to reliable distributed systems: Building high-assurance applications and cloud-hosted services*. Springer Science & Business Media, 2012, ISBN: 1447124154.
- [15] M. Paskin, C. Guestrin, and J. McFadden, “A robust architecture for distributed inference in sensor networks”, in *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, IEEE, 2005, 55–62.
- [16] “Who we are”. Internet: <https://wi-fi.org/who-we-are/our-brands> [Jul. 23, 2017].
- [17] “What Is a wireless network?: - Wi-Fi Network”. Internet: <https://cisco.com/c/en/us/solutions/small-business/resource-center/work-anywhere/wireless-network.html> [Jul. 23, 2017].

- [18] J. F. Kurose and K. W. Ross, *Computer networking: A top-down approach*. Pearson/Addison Wesley, 2008, ISBN: 9780135928790.
- [19] ITU, “ITU-T Recommendation G. 1010: End-user multimedia”, *ITU*, November, 2001.
- [20] A. F. Anta, C. Georgiou, D. R. Kowalski, J. Widmer, and E. Zavou, “Measuring the impact of adversarial errors on packet scheduling strategies”, *Journal of Scheduling*, vol. 19, no. 2, 135–152, 2016, ISSN: 0302-9743.
- [21] C. Anglano and M. Canonico, “Performance analysis of high-performance file-transfer systems for grid applications”, *Concurrency and Computation: Practice and Experience*, vol. 18, no. 8, 807–816, 2006, ISSN: 1532-0626.
- [22] W. E. Castellanos, J. C. Guerri, and P. Arce, “A QoS-aware routing protocol with adaptive feedback scheme for video streaming for mobile networks”, *Computer Communications*, vol. 77, 10–25, 2016, ISSN: 0140-3664.
- [23] S. Xu, M. Perez, K. Yang, C. Perrenot, J. Felblinger, and J. Hubert, “Determination of the latency effects on surgical performance and the acceptable latency levels in telesurgery using the dV-Trainer® simulator”, *Surgical endoscopy*, vol. 28, no. 9, 2569–2576, 2014, ISSN: 0930-2794.
- [24] K. Juvva, “Real-time systems”, 1998. Internet: https://users.ece.cmu.edu/~koopman/des_s99/real_time/index.html [May 26, 2017].
- [25] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, “A firm real-time system implementation using commercial off-the-shelf hardware and free software”, in *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No. 98TB100245)*, IEEE, 1998, 112–119.
- [26] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, “Performance analysis of high performance computing applications on the amazon web services cloud”, in *2nd IEEE international conference on cloud computing technology and science*, IEEE, 2010, 159–168.
- [27] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things”, in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, ACM, 2012, 13–16.
- [28] D. Pizzolli, G. Cossu, D. Santoro, L. Capra, C. Dupont, D. Charalampos, F. De Pellegrini, F. Antonelli, and S. Cretti, “Cloud4iot: A heterogeneous, distributed and autonomic cloud platform for the iot”, in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2016, 476–479.
- [29] W. Li and W. Shen, “Swarm behavior control of mobile multi-robots with wireless sensor networks”, *Journal of Network and Computer Applications*, vol. 34, no. 4, 1398–1407, 2011, ISSN: 1084-8045.
- [30] J. Nembrini, “Minimalist coherent swarming of wireless networked autonomous mobile robots”, PhD thesis, University of the West of England, Bristol, 2005.
- [31] S. Weber, “Design science research: Paradigm or approach?”, in *AMCIS*, 2010, p. 214.
- [32] B. J. Oates, *Researching information systems and computing*. Sage, 2005, ISBN: 9781412902236.
- [33] F. Kaboub, “Positivist paradigm”, *Encyclopaedia of Counselling*, vol. 2, no. 2, p. 343, 2008.

- [34] J. E. Sale, L. H. Lohfeld, and K. Brazil, “Revisiting the quantitative-qualitative debate: Implications for mixed-methods research”, *Quality and quantity*, vol. 36, no. 1, 43–53, 2002.
- [35] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research”, *Journal of management information systems*, vol. 24, no. 3, 45–77, 2007, ISSN: 0742-1222.
- [36] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014, ISBN: 3662524465.
- [37] R. H. Von Alan, S. T. March, J. Park, and S. Ram, “Design science in information systems research”, 1, vol. 28, Springer, 2004, 75–105.
- [38] K. Peffers, M. Rothenberger, T. Tuunanen, and R. Vaezi, “Design science research evaluation”, in *International Conference on Design Science Research in Information Systems*, Springer, 2012, 398–410.
- [39] P. Kumari and P. Kaur, “A survey of fault tolerance in cloud computing”, *Journal of King Saud University-Computer and Information Sciences*, 2018, ISSN: 1319-1578.
- [40] M. Hasan and M. S. Goraya, “Fault tolerance in cloud computing environment: A systematic survey”, *Computers in Industry*, vol. 99, 156–172, 2018, ISSN: 0166-3615.
- [41] “iStarUSA IS-700R3KP 700W 3U Redundant Power Supply”. Internet: https://bhphotovideo.com/c/product/834790-REG/iStarUSA_IS_700R3KP_IS_700R3KP_700W_3U_Redundant.html [Jun. 06, 2019].
- [42] A. S. Tanenbaum and M. Van Steen, *Distributed systems: Principles and paradigms*. Prentice-Hall, 2014, ISBN: 9781292038001.
- [43] S. Chouikhi, I. El Korbi, Y. Ghamri-Doudane, and L. A. Saidane, “A survey on fault tolerance in small and large scale wireless sensor networks”, *Computer Communications*, vol. 69, 22–37, 2015, ISSN: 0140-3664.
- [44] P. Zhang, S. Shu, and M. Zhou, “An online fault detection model and strategies based on SVM-grid in clouds”, *IEEE/CAA Journal of Automatica Sinica*, vol. 5, no. 2, 445–456, 2018, ISSN: 2329-9266.
- [45] “Fault-tolerant”, in *Merriam-Webster Inc.* 2019. Internet: <https://merriam-webster.com/dictionary/fault-tolerant> [Apr. 17, 2019].
- [46] H. D. Tu, J. Park, S. Shimamoto, and J. Kitaori, “Performance evaluation of communication system proposed for oceanic air traffic control”, in *2010 IEEE Wireless Communication and Networking Conference*, IEEE, 2010, 1–6.
- [47] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, “Proactive fault tolerance using preemptive migration”, in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, IEEE, 2009, 252–257.
- [48] A. Polze, P. Troger, and F. Salfner, “Timely virtual machine migration for pro-active fault tolerance”, in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, IEEE, 2011, 234–243.
- [49] D. Bruneo, S. Distefano, F. Longo, A. Puliafito, and M. Scarpa, “Workload-based software rejuvenation in cloud systems”, *IEEE Transactions on Computers*, vol. 62, no. 6, 1072–1085, 2013, ISSN: 0018-9340.

- [50] S. Haider, N. R. Ansari, M. Akbar, M. R. Perwez, and K. Ghori, "Fault tolerance in distributed paradigms", in *Proceedings of fifth international conference on computer communication and management*. IACSIT Press, Singapore, 2011.
- [51] C. Wills, "Session layer", 2007. Internet: <https://web.cs.wpi.edu/~cs513/s07/week11-session.pdf> [May 29, 2017].
- [52] A. Malik, J. Qadir, B. Ahmad, K.-L. A. Yau, and U. Ullah, "QoS in IEEE 802.11-based wireless networks: a contemporary review", *Journal of Network and Computer Applications*, vol. 55, 24–46, 2015, ISSN: 1084-8045.
- [53] G. Thonet, P. Allard-Jacquín, and P. Colle, "Zigbee-WiFi coexistence", *Schneider Electric White Paper and Test Report*, vol. 1, 1–38, 2008.
- [54] J.-H. Lee, S.-J. Kho, U.-J. Jang, and Y.-S. Hong, "An improved method of avoiding RF congestion in indoor environments", *Procedia Engineering*, vol. 154, 223–228, 2016, ISSN: 1877-7058.
- [55] "What does the ITU do?" Internet: <https://itu.int/en/about/Pages/whatwedo.aspx> [Jun. 20, 2017].
- [56] J. M. Gimenez-Guzman, I. Marsa-Maestre, D. Orden, E. de la Hoz, and T. Ito, "On the goodness of using orthogonal channels in WLAN IEEE 802.11 in realistic scenarios", *Wireless Communications and Mobile Computing*, vol. 2018, 2018, ISSN: 1530-8669.
- [57] "Broadband: Understanding wireless channels". Internet: <https://support.zen.co.uk/kb/Knowledgebase/Broadband-Understanding-Wireless-Channels> [Sep. 18, 2018].
- [58] J. Hintersteiner, "Channel bonding in WiFi: rules and regulations". Internet: <https://networkcomputing.com/wireless-infrastructure/channel-bonding-wifi-rules-and-regulations> [Sep. 18, 2018].
- [59] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: A simple model and its empirical validation", *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, 303–314, 1998.
- [60] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, "Improving TCP/IP performance over wireless networks", in *Proceedings of the 1st annual international conference on Mobile computing and networking*, ACM, 1995, 2–11.
- [61] "Ossification", in *Merriam-Webster.com*, 2019. Internet: <https://merriam-webster.com/dictionary/ossification> [Jun. 04, 2019].
- [62] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, *et al.*, "De-ossifying the internet transport layer: A survey and future perspectives", *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, 619–639, 2016, ISSN: 1553-877X.
- [63] L. L. Peterson and B. S. Davie, *Computer networks: A systems approach*, 1st ed. Morgan Kaufmann, 2012, p. 884, ISBN: 9780128103517.
- [64] C. E. Shannon, *A mathematical theory of communication*, 3. Wiley Online Library, 1948, vol. 27, 379–423, ISBN: 0252725484.
- [65] H. Trsek, *Isochronous wireless network for real-time communication in industrial automation*. Springer, 2016, ISBN: 9783662491577.
- [66] M. J. Flynn and M. J. Flynn, "Very high-speed computing systems", *Proceedings of the IEEE*, vol. 54, no. 12, 1901–1909, 1966.

- [67] I. Chivers and J. Sleightholme, *Introduction to programming with Fortran: With coverage of Fortran 90, 95, 2003, 2008 and 77*. Springer, 2015, 1–674, ISBN: 9783319177014. DOI: 10.1007/978-3-319-17701-4.
- [68] M. J. Flynn, “Some computer organizations and their effectiveness”, 9, vol. 100, IEEE, 1972, 948–960.
- [69] M. Otoom and J. M. Paul, “Workload mode identification for chip heterogeneous multiprocessors”, *International Journal of Parallel Programming*, vol. 40, no. 2, 184–224, 2012, ISSN: 0885-7458.
- [70] H. El-Rewini and M. Abd-El-Barr, *Advanced computer architecture and parallel processing*. John Wiley & Sons, 2005, vol. 42, ISBN: 1280254742.
- [71] N. Sadashiv and S. D. Kumar, “Cluster, grid and cloud computing: A detailed comparison”, in *2011 6th International Conference on Computer Science & Education (ICCSE)*, IEEE, 2011, 477–482.
- [72] K. Krauter, R. Buyya, and M. Maheswaran, “A taxonomy and survey of grid resource management systems for distributed computing”, *Software: Practice and Experience*, vol. 32, no. 2, 135–164, 2002, ISSN: 1877-0509.
- [73] J. Baker, “By 2020, 50% of managed APIs projected to be event-driven”. Internet: <https://realtimeapi.io/2020-50-percent-managed-apis-projected-event-driven> [Mar. 03, 2019].
- [74] Q. F. Stout, “What is parallel computing?” Internet: <https://web.eecs.umich.edu/~qstout/parallel.html> [Dec. 13, 2018].
- [75] D. Andrade, B. B. Fraguera, J. Brodman, and D. Padua, “Task-parallel versus data-parallel library-based programming in multicore systems”, in *2009 17th euromicro international conference on parallel, distributed and network-based processing*, IEEE, 2009, 101–110.
- [76] P. Pandey, H. Viswanathan, and D. Pompili, “Robust orchestration of concurrent application workflows in mobile device clouds”, *Journal of Parallel and Distributed Computing*, vol. 120, 101–114, 2018, ISSN: 0743-7315.
- [77] I. Foster, Y. Zhao, I. Raicu, and S. Lu, “Cloud computing and grid computing 360-degree compared”, in *2008 Grid Computing Environments Workshop*, IEEE, 2008, ISBN: 978-1-4244-2860-1.
- [78] “MPI forum”. Internet: <https://mpi-forum.org> [Apr. 16, 2017].
- [79] “Open source high performance computing”. Internet: <https://www.open-mpi.org> [Apr. 16, 2017].
- [80] “High-performance portable MPI”. Internet: <https://www.mpich.org> [Apr. 16, 2017].
- [81] J. Kraus, “An introduction to CUDA-aware MPI”. Internet: <https://devblogs.nvidia.com/introduction-cuda-aware-mpi> [Aug. 19, 2018].
- [82] I. Foster, “What is the grid? A three point checklist”, *GRID today*, vol. 1, no. 6, 32–36, 2002.
- [83] K. Kaur and A. K. Rai, “A comparative analysis: grid, cluster and cloud computing”, *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 3, no. 3, 5730–5734, 2014.
- [84] “Compute for science”. Internet: <https://boinc.berkeley.edu> [Dec. 11, 2018].
- [85] “About Folding@Home”. Internet: <https://foldingathome.org/about> [Mar. 12, 2017].

- [86] “Fog computing and the internet of things: Extend the cloud to where the things are”. Internet: https://cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf [Feb. 19, 2019].
- [87] “Architecture”, in *Merriam-Webster.com*, 2019. Internet: <https://merriam-webster.com/dictionary/architecture> [Jun. 13, 2019].
- [88] B. P. Douglass, *Real-time design patterns: robust scalable architecture for real-time systems*. Addison-Wesley Professional, 2003, ISBN: 0201699567.
- [89] S. Sebastian *et al.*, “Transform monolith into microservices using Docker”, in *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, IEEE, 2017, 1–5.
- [90] J.-P. Gouigoux and D. Tamzalit, “From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture”, in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, IEEE, 2017, 62–65.
- [91] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, “From monolithic to microservices: An experience report from the banking domain”, *Ieee Software*, vol. 35, no. 3, 50–55, 2018, ISSN: 0740-7459.
- [92] D. Taibi, V. Lenarduzzi, and C. Pahl, “Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation”, *IEEE Cloud Computing*, vol. 4, no. 5, 22–32, 2017.
- [93] N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, “*Microservices: Migration of a mission critical system*”, 2017.
- [94] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, “Microservices: How to make your application scale”, in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, Springer, 2017, 95–104.
- [95] N. Avery, “Journey to event driven – Part 1: Why event-first thinking changes everything”. Internet: <https://confluent.io/blog/journey-to-event-driven-part-1-why-event-first-thinking-changes-everything> [Jun. 18, 2019].
- [96] —, “Journey to event driven – Part 2: Programming models for the event-driven architecture”. Internet: <https://confluent.io/blog/journey-to-event-driven-part-2-programming-models-event-driven-architecture> [Jun. 18, 2019].
- [97] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, “The reactive manifesto”. Internet: <https://www.reactivemanifesto.org> [Oct. 26, 2019].
- [98] T. Wells, “Tom Wells - RESTful must die! Challenges building realtime AI at scale.” 2019. Internet: <https://youtube.com/watch?v=OyZks0tbD4c> [Jun. 18, 2019].
- [99] P. Di Francesco, P. Lago, and I. Malavolta, “Architecting with microservices: A systematic mapping study”, *Journal of Systems and Software*, vol. 150, no. April, 77–97, 2019, ISSN: 0164-1212. DOI: 10.1016/j.jss.2019.01.001. Internet: <https://doi.org/10.1016/j.jss.2019.01.001>.
- [100] D. Firesmith, “Virtualization via containers”. Internet: https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-containers.html [Jun. 16, 2018].
- [101] S. J. Bigelow, “What’s the difference between type 1 and type 2 hypervisors?” Internet: <https://searchservirtualization.techtarget.com/feature/Whats-the-difference-between-Type-1-and-Type-2-hypervisors> [Mar. 18, 2019].

- [102] B. Doerrfeld, “5 container alternatives to Docker”. Internet: <https://containerjournal.com/topics/container-ecosystems/5-container-alternatives-to-docker> [Apr. 27, 2019].
- [103] “What is a container?” Internet: <https://docker.com/resources/what-container> [Nov. 18, 2018].
- [104] P. Raj, J. Chelladhurai, and V. Singh, “Containerization vs. virtualization: An introduction to Docker”. Internet: <https://jaxenter.com/containerization-vs-virtualization-docker-introduction-120562.html> [Jan. 18, 2019].
- [105] “Production-grade container orchestration”. Internet: <https://kubernetes.io> [Jun. 08, 2019].
- [106] P. Heidari, Y. Lemieux, and A. Shami, “QoS assurance with light virtualization-a survey”, in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2016, 558–563.
- [107] J. Penders, L. Alboul, U. Witkowski, A. Naghsh, J. Saez-Pons, S. Herbrechtsmeier, and M. El-Habbal, “A robot swarm assisting a human fire-fighter”, *Advanced Robotics*, vol. 25, no. 1-2, 93–117, 2011, ISSN: 0169-1864.
- [108] E. Şahin, “Swarm robotics: From sources of inspiration to domains of application”, in *International workshop on swarm robotics*, 2004, 10–20.
- [109] C. J. DeHeer, “A comparison of the colony-founding potential of queens from single-and multiple-queen colonies of the fire ant”, *Animal Behaviour*, vol. 64, no. 4, 655–661, 2002, ISSN: 0003-3472.
- [110] R. Beckers, S. Goss, J.-L. Deneubourg, and J.-M. Pasteels, “Colony size, communication and ant foraging strategy”, *Psyche: A Journal of Entomology*, vol. 96, no. 3-4, 239–256, 1989.
- [111] “Robocup”. Internet: <https://www.robocup.org> [Oct. 24, 2019].
- [112] T. Wallace, “RoboCup 2017 wrap-up: Highs, lows, plenty of falls”. Internet: <https://cosmosmagazine.com/technology/robocup-2017-wrap-up-highs-lows-plenty-of-falls> [Nov. 02, 2019].
- [113] W. Vandenberghe, I. Moerman, and P. Demeester, “Adoption of vehicular ad hoc networking protocols by networked robots”, *Wireless Personal Communications*, vol. 64, no. 3, 489–522, 2012, ISSN: 0929-6212.
- [114] S. Hedberg, “Robots playing soccer? RoboCup poses a new set of AI research challenges”, *IEEE Concurrency*, vol. 5, no. 4, 5–9, Oct. 1997, ISSN: 1092-3063.
- [115] I. Jawhar, N. Mohamed, J. Wu, and J. Al-Jaroodi, “Networking of multi-robot systems: Architectures and requirements”, *Journal of Sensor and Actuator Networks*, vol. 7, no. 4, p. 52, 2018.
- [116] S. Luo, W. Yao, Q. Yu, J. Xiao, H. Lu, and Z. Zhou, “Object detection based on GPU parallel computing for RoboCup middle size league”, in *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, IEEE, 2017, 86–91.
- [117] “Power consumption benchmarks”. Internet: <https://pidramble.com/wiki/benchmarks/power-consumption> [Jun. 20, 2019].
- [118] T. R. Burchfield, S. Venkatesan, and D. Weiner, “Maximizing Throughput in ZigBee Wireless Networks through Analysis, Simulations and Implementations”, *Proc. Int. Workshop Localized Algor. Protocols WSNs*, pp. 15–29, 2007. Internet: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.129.8350>.

- [119] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne, “Understanding the Limits of LoRaWAN”, *IEEE Communications Magazine*, vol. 55, no. 9, pp. 34–40, 2017, ISSN: 01636804. DOI: 10.1109/MCOM.2017.1600613. arXiv: 1607.08011.
- [120] J. Pojda, A. Wolff, M. Sbeiti, and C. Wietfeld, “Performance analysis of mesh routing protocols for UAV swarming applications”, in *2011 8th International Symposium on Wireless Communication Systems*, IEEE, 2011, 317–321.
- [121] D. Qiao, S. Choi, and K. G. Shin, “Goodput analysis and link adaptation for IEEE 802.11 a wireless LANs”, *IEEE transactions on Mobile Computing*, vol. 1, no. 4, 278–292, 2002.
- [122] M. Jünemann, “802.11s wireless mesh network”. Internet: https://mjuenema.github.io/80211s_wireless_mesh [Apr. 17, 2018].
- [123] M. de Bayser and R. Cerqueira, “Integrating MPI with Docker for HPC”, in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, 2017, 259–265.
- [124] University Ohio State and K Dhabaleswar, “MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE”. Internet: <http://www.mvapich.cse.ohio-state.edu> [Jul. 17, 2017].
- [125] N. Nguyen and D. Bein, “Distributed MPI cluster with Docker Swarm mode”, in *2017 IEEE 7th annual computing and communication workshop and conference (ccwc)*, IEEE, 2017, 1–7.
- [126] “JSON-RPC 2.0 specification”. Internet: <https://jsonrpc.org/specification> [Aug. 18, 2018].
- [127] “Promise”. Internet: <https://javascript.info/promise-basics> [Jun. 30, 2019].
- [128] N. Narkhede, “Exactly-once semantics are possible: Here’s how Kafka does it”. Internet: <https://confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it> [Apr. 24, 2019].
- [129] “Kafka - Message delivery semantics”. Internet: <https://kafka.apache.org/documentation/#semantics> [Mar. 16, 2019].
- [130] J.-P. Azar, “Kafka Architecture”. Internet: <https://dzone.com/articles/kafka-architecture> [Jul. 15, 2019].
- [131] “Introduction”. Internet: <https://kafka.apache.org/intro> [Feb. 12, 2019].
- [132] “Cloud Native Interactive Landscape”. Internet: <https://landscape.cncf.io> [Apr. 17, 2019].
- [133] “About NATS”. Internet: <https://nats.io/about> [Apr. 04, 2019].
- [134] “NATS streaming concepts”. Internet: https://nats-io.github.io/docs/nats_streaming/intro.html [May 18, 2019].
- [135] “YAML 1.2”. Internet: <https://yaml.org> [Jun. 14, 2019].
- [136] “Raft consensus in Swarm mode”. Internet: <https://docs.docker.com/engine/swarm/raft> [Oct. 16, 2019].
- [137] O. Khan, M. Jameel, and A. Shafi, “High performance message-passing InfiniBand communication device for Java HPC”, *Procedia Computer Science*, vol. 29, 1468–1479, 2014, ISSN: 1877-0509.
- [138] D. J. Kerbyson, K. J. Barker, A. Vishnu, and A. Hoisie, “A performance comparison of current HPC systems: Blue Gene/Q, Cray XE6 and InfiniBand systems”, *Future Generation Computer Systems*, vol. 30, 291–304, 2014, ISSN: 0167-739X.

- [139] L. Frenzel, “What’s The Difference Between EM Near Field And Far Field?”, 2012. Internet: <https://www.electronicdesign.com/markets/energy/article/21801976/whats-the-difference-between-em-near-field-and-far-field> [Feb. 03, 2020].
- [140] Occupation Safety and Health Administration, “Electromagnetic Radiation Field Memo”, 1990. Internet: https://www.osha.gov/SLTC/radiofrequencyradiation/electromagnetic{__}fieldmemo/electromagnetic.html [Feb. 03, 2020].
- [141] “B.A.T.M.A.N. advanced”. Internet: <https://open-mesh.org/projects/batman-adv/wiki/Wiki> [Nov. 15, 2018].
- [142] E. Au, “New initiatives in IEEE 802.11 for V2X, broadcast services and full duplex”. Internet: <https://vtsociety.org/2018/05/new-initiatives-in-ieee-802-11-for-v2x-broadcast-services-and-full-duplex> [Jul. 29, 2019].
- [143] “Slow consumers”. Internet: <https://docs.nats.io/developing-with-nats/events/slow> [Jul. 14, 2019].
- [144] R. Zwetsloot, “Raspberry Pi 4 specs and benchmarks”. Internet: <https://magpi.raspberrypi.org/articles/raspberry-pi-4-specs-benchmarks> [Oct. 25, 2019].
- [145] M. Edenhill, “librdkafka - the Apache Kafka C/C++ client library”. Internet: <https://github.com/edenhill/librdkafka> [Mar. 22, 2019].
- [146] “node-rdkafka - Node.js wrapper for Kafka C/C++ library”. Internet: <https://github.com/Blizzard/node-rdkafka> [Mar. 23, 2019].

Appendix A

Additional figures and tables

In this appendix, additional information related to the literature review presented in Chapter 2 will be presented. The additional information supports the narrative of the chapter, but do not need to be read side-by-side. The content of this appendix has been placed here in order to improve the focus of the chapter.

A.1 Wireless Networking

The figures within this section are relevant to Section 2.2 on page 21. Figure A.1 shows a WiFi analyser mobile application and visually represents one of the challenges of wireless networking presented in Section 2.2.3 on page 23.

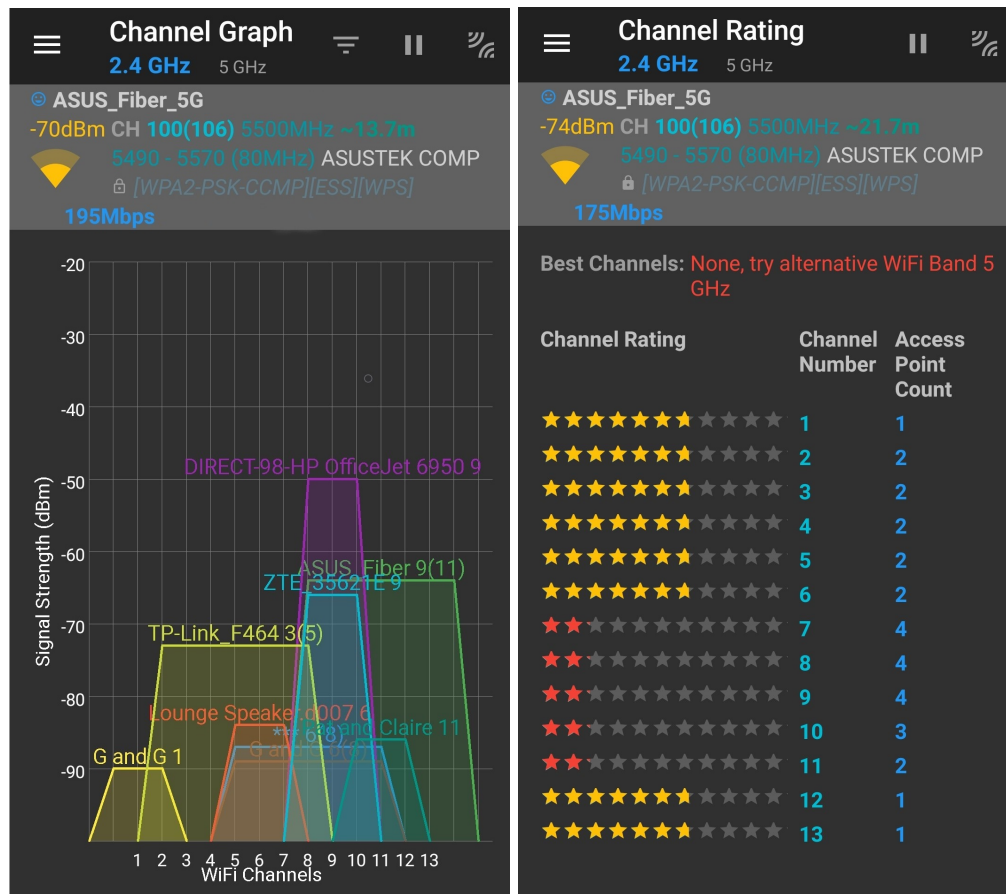


Figure A.1 A Wi-Fi analyser mobile app can visualise Wi-Fi channels and recommend the “best” channels to use. The app visualises the crowded nature of wireless channels in highly populated areas.

Figure A.2 on the following page presents Papastergiou’s [62] potential solution to the transport protocol ossification problem which affects wireless networks as presented in Section 2.2.4 on page 26.

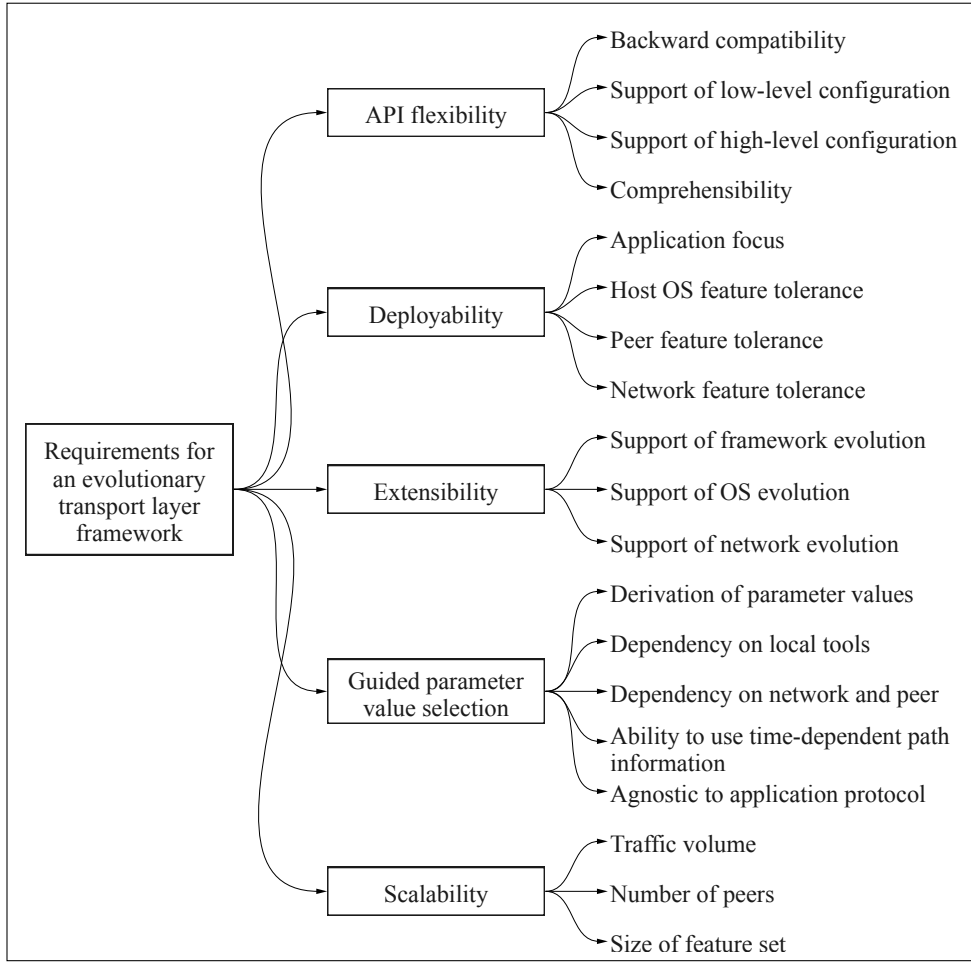


Figure A.2 Identified requirements for an evolutionary transport-layer framework as a solution to the ossified transport protocols in use today. Figure adapted from [62].

A.2 Distributed computing

The figures within this section are relevant to Section 2.3 on page 30. Table A.1 shows how different kinds of devices all leverage multiple cores for processing.

Device	Processor	Cores / Threads (Arch.)	Freq. (Ghz)
Asus AC2900 Router	BCM4906	2 / 2 (ARM)	1.8
Raspberry Pi 3 Model B+	BCM2837	4 / 4 (ARM)	1.2
Samsung Galaxy S9	Exynos 9810	8 / 8 (ARM big.LITTLE)	1.7 / 2.7
Laptops	i7-8650U	4 / 8 (x86/AMD64)	4.2
Workstations	2990WX	32 / 64 (x86/AMD64)	4.2
HPC Servers	TU102	4608 / 4608 (CUDA)	1.5

Table A.1 The number of processing cores in various computing devices.

Table A.2 presents an in-depth comparison between clusters, grids and cloud computing.

	Clusters	Grids	Clouds
SLA	Limited	Yes	Yes
Allocation	Centralised	Decentralised	Both
Resource Handling	Centralised	Distributed	Both
Loose coupling	No	Both	Yes
Reliability	No	Half	Full
Protocols/API	MPI, Parallel Virtual	MPI, MPICH-G, GIS,GRAM	TCP/IP,SOAP, REST,AJAX
Security	Yes	Half	No
User friendliness	No	Half	Yes
Virtualisation	Half	Half	Yes
Interoperability	Yes	Yes	Half
Standardised	Yes	Yes	No
Business Model	No	No	Yes
Task Size	Single large	Single large	Small & medium
SOA	No	Yes	Yes
Multi-tenancy	No	Yes	Yes
System Performance	Improves	Improves	Improves
Self-service	No	Yes	Yes
Computation service	Computing	Max. Computing	On demand
Heterogeneity	No	Yes	Yes
Scalable	No	Half	Yes
Inexpensive	No	No	Yes
Data Locality Exploited	No	No	Yes
Application	HPC, HTC	HPC, HTC, Batch	SME interactive apps
Switching cost	Low	Low	High
Value Added Services	No	Half	Yes

Table A.2 In-depth comparison between Clusters, Grids and Clouds [71]. The categories most relevant to this research are reliability and loose coupling.

Figures A.3 to A.4 on the following page present representative task that leverage task parallelism and data parallelism. A third task which leverages a combination of both approaches to parallelism is presented in Figure 2.10 on page 34.

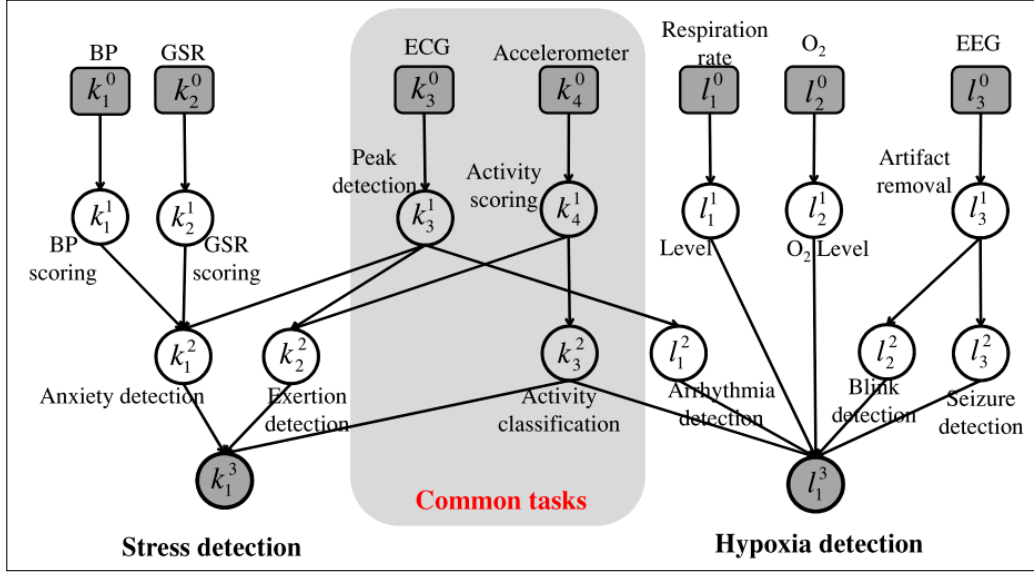


Figure A.3 Workflow from the healthcare domain [76]. The implementation is an example of a purely task parallel workflow.

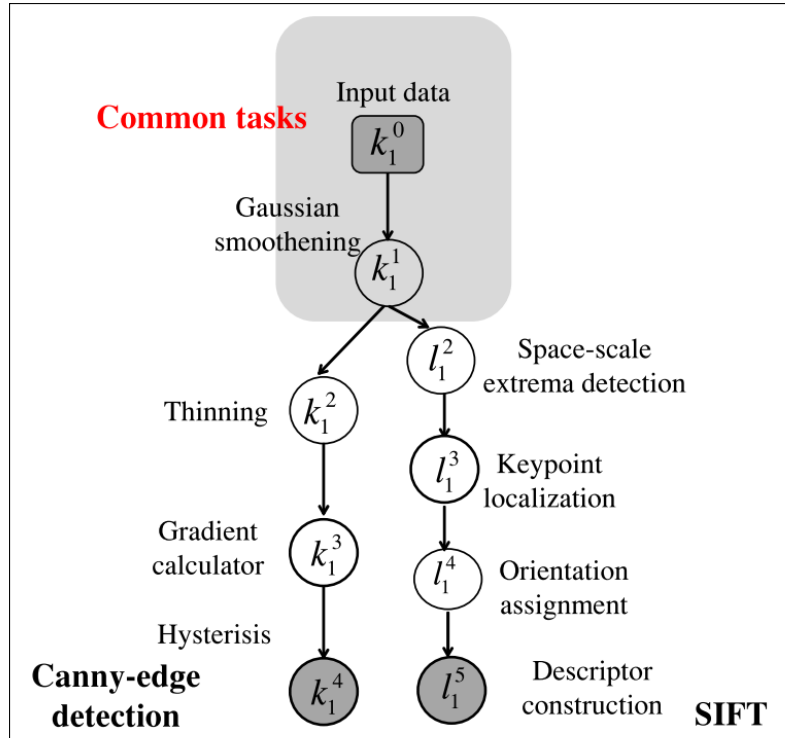


Figure A.4 Workflow from the computer vision domain [76]. The implementation is an example of a purely data parallel workflow.

A.3 System architecture

Figures A.5 to A.6 on pages 127–128 present a case study of a migration from a monolithic core banking system to one implemented using microservices. The monolithic implementation is characterized by tight coupling of components that only function as a collection whereas the microservice implementations allows looser coupling with increased fault tolerance.

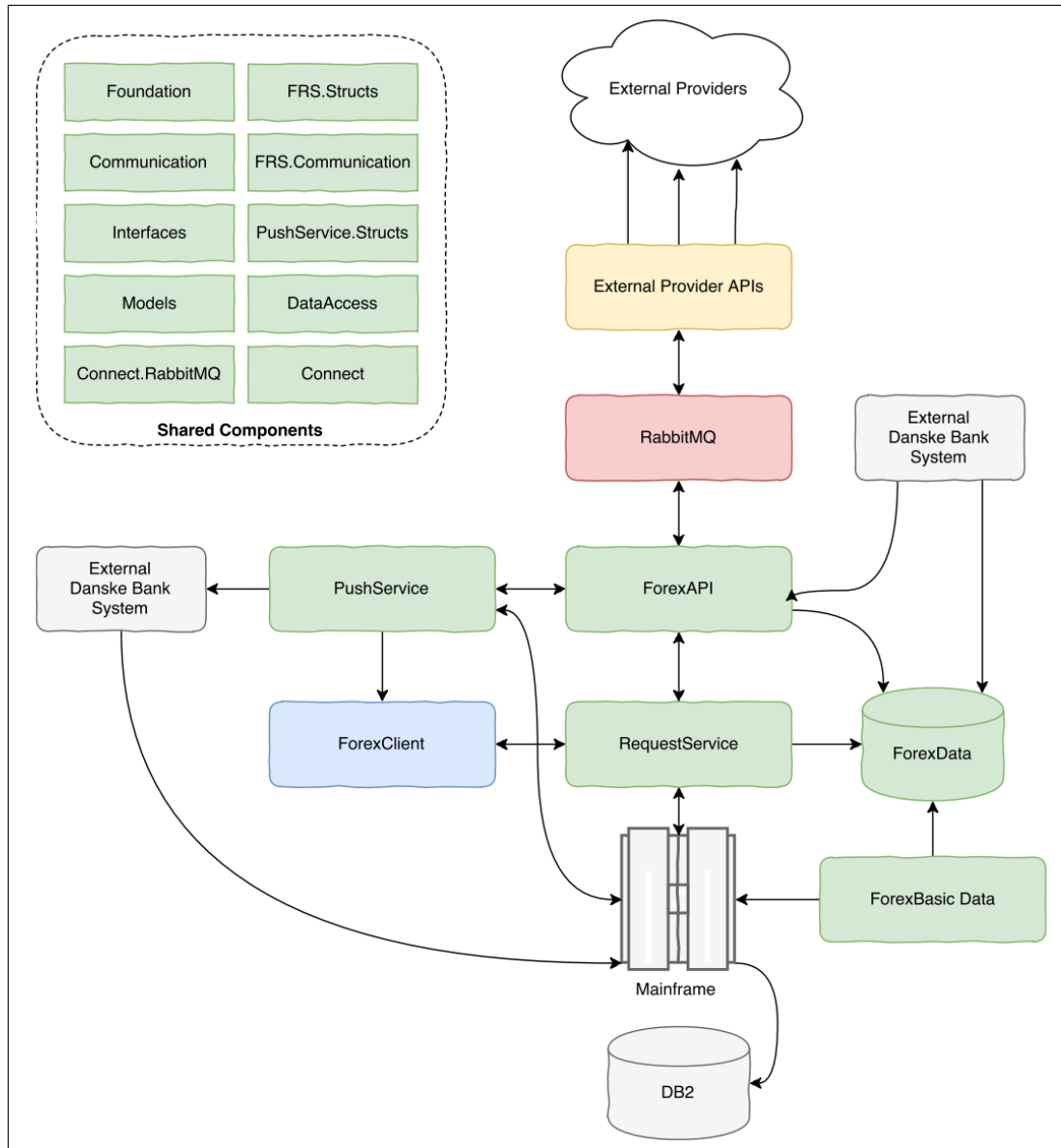


Figure A.5 Architecture overview of the FX Core system used at a major bank in Denmark, implemented using a legacy monolith architecture [93].

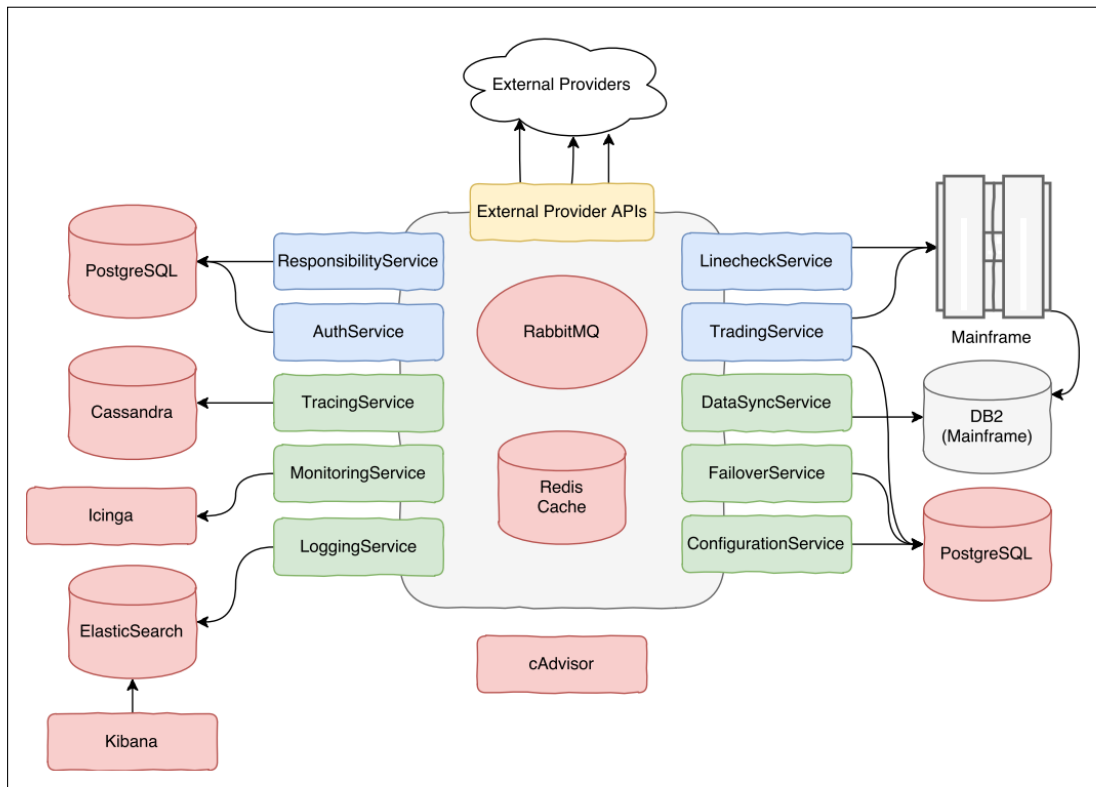


Figure A.6 Architecture overview of the FX Core system used at a major bank in Denmark, implemented using a microservice architecture [93].

Appendix B

Artefact Setup

In this appendix, additional information is included that will aid in the recreation of the artefact. The information is more technical and presented to complement the information presented in the relevant sections. Section B.1 includes information relevant to the platform supporting the application. Section B.2 includes information relevant to the application itself.

B.1 Platform configuration

In this section, detailed information is presented on the design and configuration of the platform. Starting at the bottom layer of the artefact, the hardware choices are presented in Section B.1.1. The setup and configuration of the operating system that runs on the hardware are presented in Section B.1.2. The mesh networking setup used to connect multiple nodes is presented in Section B.1.3. This section concludes with the setup of the container orchestration solution in Section B.1.4.

B.1.1 Hardware choices

The hardware choices for the artefact include:

- 3x Raspberry Pi 3 Model B+ single-board computer (SBC), hereafter referred to simply as Pi or node;
- 4x BlueWay USB wireless network adapters using the Ralink RT3070 chipset. One was used on each node, and another was needed for the desktop system to connect to the mesh; and
- Raspberry Pi "Cluster" case and aluminium heatsinks for improved cooling (see Figure B.1 on the following page). Improving the cooling successfully prevented throttling of the processors which could lead to inconsistent results.

The Raspberry Pi 3 Model B+ was the latest model available at the time of purchase. The model has been superseded by the Raspberry Pi 4 Model B which has a significantly faster processor and a larger memory capacity of up to 4GB. Revisiting this research with the more powerful SBC is included as a recommendation for future work in Section 5.3.

B.1.2 OS setup and configuration

The operating system used on each Raspberry Pi was the 6.17 release of the DietPi¹ image. DietPi is significantly more lightweight than the official distribution, Raspbian. DietPi also includes various configuration tools that simplify setup. Balena Etcher ² was used to flash the DietPi image to the memory card used by the nodes. On first boot, the following changes were made to each Pi, using the *dietpi-config* tool:

¹<https://dietpi.com/>

²balena.io/etcher



Figure B.1 Improved cooling of Raspberry Pi's using an open airflow case and aluminium heatsinks.

1. Changed the default password;
2. Reduced GPU RAM allocation to the minimum of 16MB;
3. Set CPU governor to performance;
4. Set system locale to EN_ZA.UTF8;
5. Set time zone to Africa/Johannesburg;
6. Set keyboard layout to US;
7. Changed the system hostname to pi-1, pi-2 and pi-3 respectively;
8. Disabled the on-board Wi-Fi adapter, forcing the Pi to transmit using the USB adapters; and
9. Enabled automatic console login.

After these changes were made, the Pi was rebooted. At this point, an internet connection is required and provided using an ethernet cable. Script B.1 shows the commands used to install the required dependencies. The script also updates any out-of-date packages included with the minimal DietPi image and installs useful network analysis tools such as *nmap*, *bmon* and *wavemon*.

```

1  ## Update repositories and upgrade any packages with newer versions ##
2  sudo apt update
3  sudo apt upgrade -y
4
5  ## Install dependencies & network tools ##
6  sudo apt install --no-install-recommends -y          \
7      nmap insserv bmon cpufrequtils wavemon          \
8      build-essential curl gdebi git htop libnl-3-dev  \
9      libnl-genl-3-dev net-tools pkg-config wireless-tools \
10     firmware-linux-nonfree firmware-linux
```

Script B.1 Dependency installation on DietPi. Some dependencies are already included in Raspbian, but have to be manually installed when using the light DietPi image.

B.1.3 Mesh networking setup

The next step is to install the *batctl* BATMAN control interface, which is required for configuring a mesh network using the *batman-adv* kernel module to facilitate routing. The *batctl* utility is not available from typical repositories, so the source repository is cloned and compiled. Script B.2 presents the procedure.

```
1  ## Clone BATMANCTL source, compile and install ##
2  cd ~
3  git clone https://git.open-mesh.org/batctl.git --depth 1
4  cd batctl
5  sudo make -j 4
6  sudo make install
7  cd ~
8  sudo rm -R ~/batctl
9  sudo reboot
```

Script B.2 Raspberry Pi BATMAN installation.

With *batctl* installed, everything is ready to start a mesh network. To automate the process for each Pi, a script was developed that will be used to start the mesh network using the correct adapter at boot. Script B.3 on the following page performs several functions in order to do establish the mesh:

1. *Lines 3-10*: Reads the hostname of the device that the script is running on and sets a *MESHIP* variable that will be used to identify the device in the mesh network;
2. *Line 12*: Extracts the name of the mesh-enabled NIC based on its MAC address. All of the BlueWay adapters used had a MAC address starting with *00:0f:00*, so it is possible to distinguish all of the external adapters from the built-in adapters in this way;
3. *Line 21*: Activates the *batman-adv* kernel module already included in the running Linux kernel;
4. *Line 22-24*: Disables power-saving mode, turns the adapter off and sets the MTU value;
5. *Line 25-28*: Enables ad-hoc mode, sets the mesh name to *raspi-mesh*, allows the adapter to connect to any mesh neighbour and sets the channel;
6. *Line 29*: Turns the adapter on;
7. *Line 30-31*: Removes and re-adds the adapter from *batcl*;
8. *Line 32-33*: Turns on the virtual *bat0* interface and assigns it the *MESHIP* variable IP;
9. *Line 35*: Triggers a watch on the *batcl n* command, which will print the known neighbours. It usually takes a few seconds for all the mesh adapters to show up.

```

1  #!/bin/bash
2  ## Each adapter gets an IP/CIDR based on hostname ##
3  case $HOSTNAME in
4      ("TinyTower") MESHIP="10.0.0.1/24" ;;
5      ("pi-console") MESHIP="10.0.0.10/24" ;;
6      ("pi-1") MESHIP="10.0.0.11/24" ;;
7      ("pi-2") MESHIP="10.0.0.12/24" ;;
8      ("pi-3") MESHIP="10.0.0.13/24" ;;
9      (*)  echo "No recognized hostname, exiting." && exit 1
10 esac
11 ##Identify mesh adapter by MAC Address ##
12 ADAPTERNAME=$(ip -o link | awk '$(NF-2) ~ "00:0f:00" {print $2}' | tr -d ':' | grep
    w)
13 if [ -z "$ADAPTERNAME" ]
14 then
15     echo "No adapter found!"
16     exit 1
17 else
18     echo "Using $ADAPTERNAME"
19 fi
20 echo "This node will be accessible on the mesh network @ " $MESHIP
21 sudo modprobe batman-adv                #enable kernel module
22 sudo iw $ADAPTERNAME set power_save off  #disable adapter powersaving
23 sudo ip link set $ADAPTERNAME down      #turn off adapter
24 sudo ifconfig $ADAPTERNAME mtu 1500     #set MTU
25 sudo iwconfig $ADAPTERNAME mode ad-hoc  #set adapter to mesh mode
26 sudo iwconfig $ADAPTERNAME essid raspi-mesh #set mesh identifier
27 sudo iwconfig $ADAPTERNAME ap any       #connect to any neighbor
28 sudo iwconfig $ADAPTERNAME channel 1    #set adapter channel
29 sudo ip link set $ADAPTERNAME up        #turn on adapter
30 sudo batctl if del $ADAPTERNAME          #remove adapter from batctl
31 sudo batctl if add $ADAPTERNAME         #add adapter to batctl
32 sudo ifconfig bat0 up                   #enable bat0 virtual interface
33 sudo ifconfig bat0 $MESHIP              #set designated bat0 IP
34
35 sudo watch batctl n

```

Script B.3 Bash script that initialises the mesh network on each node.

This script is made executable and place in `/root/startMesh.sh`. Running this script starts the mesh, but it is worthwhile to automate the running of this script every time the Pi is restarted. To run this script automatically after boot, it is registered using `crontab -e` as presented in Script B.4 on the following page.

```

1  #add this line to the bottom of the crontab file:
2  @reboot    /root/startMesh.sh

```

Script B.4 Entry in the *crontab* that runs Script B.3 at every boot. This allows a node that restarts to rejoin the mesh network automatically.

From this point onward, the script will execute after the boot process finishes, persisting the mesh network across node reboots. Sharing internet access across the mesh is possible, but was not used in this research. For more information, refer to the documentation³.

B.1.4 Docker installation and Swarm configuration

Script B.5 presents the procedure used to install the Community Edition (CE) version of Docker on the Raspberry Pi. The same process could be followed on an x86-based machine running Linux. Lines 2 and 3 installs Docker using the *convenience script*⁴. After the install, any non-root user will not have access to use Docker yet. Lines 5-6 rectify this by adding the current user to the *docker* group and rebooting the node.

```

1  ## Install docker: ##
2  curl -fsSL https://get.docker.com -o get-docker.sh
3  sudo sh get-docker.sh && rm get-docker.sh
4
5  sudo usermod -aG docker $USER
6  sudo reboot

```

Script B.5 Raspberry Pi Docker installation. For the current non-root user to access Docker, it should be added to the *docker* group and the node should be restarted.

With all three nodes running Docker, the next step is to enable the Swarm mode to facilitate container orchestration. Script B.6 presents the procedure.

```

1  ## On the manager node: ##
2  docker swarm init --advertise-addr <IP of interface to use>
3  ## this will generate a command to run on all the worker nodes e.g.
4  docker swarm join --token <secure token> <provided IP>:2377
5  ## Once each worker node has joined the Swarm, from the manager node:
6  docker node ls

```

Script B.6 Docker Swarm configuration.

In line 2, the Swarm is initialised. When the node has multiple network adapters, as in the case of the platform, the *advertise-address* parameter is specified. For this platform, node *pi-1* was selected as the manager, and so the provided address was its mesh network address (10.0.0.11). Running

³<https://www.open-mesh.org/projects/batman-adv/wiki/Gateways>

⁴<https://docs.docker.com/install/linux/docker-ce/ubuntu/#install-using-the-convenience-script>

this command will print out another command that resembles line 4. Run the provided command on every node follower node. Once all the nodes have joined, run the command presented in line 6. The output should resemble Figure B.2.

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS	ENGINE VERSION
r10jh48vdf9y7p4cdgvgy8u7l *	pi-1	Ready	Active	Leader	18.09.0
9uh93ej0gonivdvzipvb0oo11	pi-2	Ready	Active		18.09.0
qxnlswfnbp159a9ydaqcz1qt3	pi-3	Ready	Active		18.09.0

Figure B.2 Functional Docker swarm on a mesh network.

This concludes the configuration of the platform. A functional container orchestration environment has been created while using mesh networking on the Raspberry Pi nodes.

B.2 Application configuration

In this section, the application-level configuration used in the artefact iterations is included. The first three iterations of the artefact (MPICH, WebSockets and RPC) were not complex enough to warrant detailed information into their configuration. All the required information for running these iterations are included in the git repository⁵. The event-streaming iterations (Apache Kafka and NATS) are complex and therefore are included in this appendix. Section B.2.1 includes information on the Kafka configuration used in Section 3.4.4.1.

B.2.1 Kafka configuration

Apache Kafka includes many configuration options; refer to the Kafka documentation⁶ for a complete list. In this research, only a small portion was used.

Leader election can be configured to prioritise either consistency or availability. The default scheme prioritises consistency and involves one of the other replicated brokers becoming the new leader if a leader fails or is taken down. Upon the return of an ex-leader, the leader partitions synchronised to it and it resumes being the leader. Availability can be promoted by allowing an out-of-sync replica to become the new leader. This would be done to maintain high read availability of its partitions, but messages that have not been synchronised to other brokers are lost. Due to the nature of the drone swarm application, availability was prioritised.

Replication factor is set on a topic and defines how many brokers need to replicate an event. The default value is 1, but this was increased to 3 (the number of nodes in the cluster) so that every event is replicated to all the other brokers.

Event acknowledgements defines the requirements that a producer has for the broker. The options are: all, 0 and 1 and define the number of brokers that need to acknowledge reception of the event before the producer considers the event processed. Setting the option to 0 reduces the transmission overhead, but nullifies any retries setting and can cause events to be lost. Setting

⁵<https://github.com/prossouw79/research-artefact>

⁶<https://kafka.apache.org/documentation/>

the option to 1 ensures that at least one broker has acknowledged the event and the producer can continue while the broker replicates the event to others in the cluster. If the message is not acknowledged within a time-out, the event will be published again, based on the retries configuration. For the drone swarm simulation application, the setting was set to *acks=all*.

Minimum in-sync replicas is closely related to the event acknowledgements setting. If a producer requires acknowledgements from all brokers, this setting specifies the minimum number of brokers that need to acknowledge the event before it is considered a successful write operation. If this number cannot be met, the producer raises an exception.

The typical configuration was used for the drone swarm simulation application: the topics were created with a replication factor of 3, event acknowledgements were set to all, and the minimum in-sync replicas setting was set to 2. In this configuration, the majority of the cluster needed to have acknowledged the message before it was considered a successful write operation. In order to get both Kafka and Zookeeper running on the Raspberry Pi's, their memory footprint needed to be controlled. Both Kafka brokers and their required Zookeeper instances run on the Java Virtual Machine (JVM) and are memory intensive. The Zookeeper instances will run on the Raspberry Pi machines using its default configuration. Unfortunately, Kafka brokers will not successfully start on a Raspberry Pi 3B+ using the default configuration, producing a *java.lang.OutOfMemoryError: Java heap space* error. This can be addressed by setting environment variables, as shown in Script B.7

```
1 export KAFKA_HEAP_OPTS="-Xms128m -Xmx512m"
2 export JVMFLAGS="-Xms128m -Xmx768m"
```

Script B.7 Environment variables required to run Apache Kafka and Zookeeper on a Raspberry Pi.

The first option limits the Kafka broker to use between 128MB and 512MB of available memory, while the second option limits the memory usage of the JVM in general. Note that the Raspberry Pi 3 Model B+ only has 1024MB of memory, so dedicating up to half of it to the message broker leaves little for the operating system, Zookeeper, and the stream processors that implement the desired functionality. The libraries used for the development of these stream operators are presented next.

B.2.2 Kafka stream operator development in Node.js

Kafka stream operators are typically developed in Java⁷ or Kotlin⁸ due to the first-class-citizen support for these languages. However, an open-source library called *librdkafka* is available. The *librdkafka* library is an open-source C implementation of the Kafka protocol [145].

The creator of the library claims producers and consumers implemented using this library can process 1 million messages/s and 3 million messages/s respectively. Several other libraries are built on *librdkafka*, such as the *node-rdkafka* [146] library which wraps the functionality provided by

⁷<https://www.java.com/en/>

⁸<https://kotlinlang.org/>

librdkafka in a package usable in the Node.js JavaScript runtime. This was useful since both the preceding WebSocket and RPC implementations were developed in Javascript to run on the Node.js runtime.

The development of stream operators is quite simple. Example stream operators implemented using *node-rdkafka* are available on the git repository for producers⁹ and consumers¹⁰. Using Apache Kafka as the message brokering solution was abandoned due its high memory usage in a memory-constrained platform. NATS Streaming provided an excellent alternative, and its configuration is presented next.

B.2.3 NATS configuration

The configuration of NATS for the platform will be presented in this section. A comprehensive list of configuration options is presented in the online documentation for NATS Server¹¹ and NATS Streaming Server¹². The relevant configuration options used for the platform are summarised in Table B.1.

Parameter	Description
-store	Store type: MEMORY FILE SQL (default: MEMORY)
-max_bytes	Max messages total size sync after every write to the replication log and message store per channel (0 for unlimited)
-max_msgs	Max number of messages per channel (0 for unlimited)
-max_age	Max duration a message can be stored (0s for unlimited)
-clustered	Run the server in a clustered configuration (default: false)
-cluster_log_cache_size	Size of log entries in cache to reduce disk IO (default: 512)

Table B.1 Notable command-line parameters presented by the NATS Streaming Server [133].

To support the drone swarm application, in-memory storage was used as event persistence was not required for the application. The servers were run in their clustered configuration to allow for high availability. The other settings were left at their default values.

In the final implementation, the combination of NATS with NATS Streaming was run in an orchestrated environment. The next section details the application deployment, which includes NATS, NATS Streaming and the various stream operators.

B.2.4 Application deployment

The deployment of the application depicted in Figure 3.21 on page 82 will be presented in this section. The deployment is defined in a *docker-compose.yml* file that includes the definition of the

⁹<https://github.com/Blizzard/node-rdkafka/blob/master/examples/producer.md>

¹⁰<https://github.com/Blizzard/node-rdkafka/blob/master/examples/consumer.md>

¹¹<https://docs.nats.io/nats-server/configuration>

¹²<https://docs.nats.io/nats-streaming-server/configuring>

virtual network and services. Scripts B.8 to B.16 on pages 137–141 present regions of this file and will be described individually.

```
1  version: "3.3"
2
3  networks:
4    main:
5      driver: overlay
6      ipam:
7        config:
8          - subnet: 192.168.5.0/24
9      driver: default
```

Script B.8 Application stack definition in docker-compose YAML: 1 of 1.

Script B.8 presents the docker-compose file-format definition on line 1. Version 3.3 is supported by Docker Engine versions 17.06.0 and enables this specification to be interpreted correctly in future versions of Docker. Line 3 starts the specification of a VXLAN called *main* to which the container that represents services can be connected. The default driver is the *bridge* driver, and it should be used when deploying the compose file on a single host that is not configured in Swarm mode. When Swarm mode is enabled to facilitate multi-host orchestration, the *overlay* network driver is used. Line 6 indicates the definition of the *ipam* (IP Address Management) and defines a subnet starting at 192.168.5.0 with a CIDR block of 24 which will give this VXLAN 256 IP addresses. The range of IP addresses that can be assigned containers will be 192.168.5.1 to 192.168.5.254. The driver is kept default, but alternative drivers could be used to change the way IP addresses are managed in this VXLAN.

```
10 services:
11   nats-1:
12     command:
13       - "-D"
14       - "-p"
15       - "4222"
16       - "-cluster"
17       - "nats://192.168.5.10:6222"
18     image: "nats:1.3.0"
19     networks:
20       main:
21         ipv4_address: "192.168.5.10"
22     ports:
23       - "14222:4222"
```

Script B.9 Application stack definition in docker-compose YAML: 2 of 11.

Script B.9 continues and starts the services definition on line 10. Line 11 defines the first service called *nats-1* and lines 12-17 specify the parameters provided to the entry point executable specified in the Docker file of the image referenced on line 18. Lines 19-21 connect this service to the network

defined in Script B.8 on the previous page with a specified IP. If the *ipv4_address* definition on line 21 is omitted, the container will receive an address from DHCP and containers can use service discovery functionality provided by Docker Swarm in order to resolve the IP to this service container. Lines 22-23 enable port-forwarding from the 4222 network port of the NATS process to 14222 on the container host.

Scripts B.10 to B.11 on pages 138–139 define the other two NATS services (*nats-2* and *nats-3*) with their appropriate command parameters, network IP and port-forwarding configuration. These services use the same NATS container image but have to be specified as separate service definitions because the command parameters are different. To run a service with multiple instances of the same container with an identical configuration, the number of *replicas* should be defined. In this case, neither IP addresses nor port-forwarding information will be defined, and service discovery functionality will be required. Docker Swarm will use an Ingress load balancer to redirect requests to container instances of the replicated service.

```

24  nats-2:
25    command:
26      - "-D"
27      - "-p"
28      - "4222"
29      - "-cluster"
30      - "nats://192.168.5.20:6222"
31      - "-routes"
32      - "nats://192.168.5.10:6222"
33    image: "nats:1.3.0"
34    networks:
35      main:
36        ipv4_address: "192.168.5.20"
37    ports:
38      - "24222:4222"

```

Script B.10 Application stack definition in docker-compose YAML: 3 of 11.

Script B.12 on the following page presents the definition of the NATS Streaming instance that provides event persistence to the cluster of NATS message brokers. Its definition is similar to the service definitions of NATS with some key differences:

- it uses a different container image (line 55);
- it uses a variant of the command definition and provides different parameters for the NATS Streaming executable (line 59);
- it exposes an additional port 8222 for monitoring (line 62);
- the defined ports are only exposed internally, as there is no port-forwarding specified (lines 61-62); and
- the service depends on the *nats-1* service and will only be provisioned once that service has been provisioned (line 63-64).

```

39  nats-3:
40    command:
41      - "-D"
42      - "-p"
43      - "4222"
44      - "-cluster"
45      - "nats://192.168.5.30:6222"
46      - "-routes"
47      - "nats://192.168.5.10:6222"
48    image: "nats:1.3.0"
49    networks:
50      main:
51        ipv4_address: "192.168.5.30"
52    ports:
53      - "34222:4222"

```

Script B.11 Application stack definition in docker-compose YAML: 4 of 11.

```

54  nats-streaming-1:
55    image: "nats-streaming"
56    networks:
57      main:
58        ipv4_address: "192.168.5.40"
59    command: -ns nats://192.168.5.10:4222 -cid drone-simulation-cluster
60    ports:
61      - "4222"
62      - "8222"
63    depends_on:
64      - nats-1

```

Script B.12 Application stack definition in docker-compose YAML: 5 of 11.

Script B.13 on the following page presents the definition of three services that use the *positionUpdate* container image. The definition of this service is similar to previous service definitions but includes the use of the *deploy* specification. The settings of interest are the placement constraints for each service, presented in lines 74-76 in Script B.13, lines 86-88 in Script B.14 and lines 98-100 in Script B.15. The placement of the NATS and NATS Streaming services do not matter. In contrast, the placement of the *positionUpdate* services matter as each instance represents the drone in the drone swarm application.

There can only be one instance of this application, and it has to run on a specified node. The placement constraints allow the definition of services that are forced to run on specific hosts. The nodes are identified by their host name, e.g. *pi-1*, *pi-2* or *pi-3*. It is also possible to constrain the placement of services by other characteristics, such as the node's role. For a complete list of deployment constraints, refer to the Docker service documentation¹³. For each of the nodes, a

¹³https://docs.docker.com/engine/reference/commandline/service_create/#specify-service-constraints---constraint

```
65  nats-processor-position-update-1:
66    image: "pietersynthesis/nats-swarm-positionupdate:armv7l"
67    networks:
68      - main
69    depends_on:
70      - nats-streaming-1
71    deploy:
72      mode: replicated
73      replicas: 1
74      placement:
75        constraints:
76          - node.hostname == pi-1
```

Script B.13 Application stack definition in docker-compose YAML: 6 of 11.

corresponding *positionUpdate* service is created that is forced to run on that specific node. These service definitions are included in Scripts B.14 to B.15 on pages 140–141

```
77  nats-processor-position-update-2:
78    image: "pietersynthesis/nats-swarm-positionupdate:armv7l"
79    networks:
80      - main
81    depends_on:
82      - nats-streaming-1
83    deploy:
84      mode: replicated
85      replicas: 1
86      placement:
87        constraints:
88          - node.hostname == pi-2
```

Script B.14 Application stack definition in docker-compose YAML: 7 of 11.

Script B.16 on the following page presents the definition of the *distanceCalculator* service. This service, as its name suggests, is responsible for calculating the distances between all pairs of drones. Similar to the *positionUpdate* service, this service also needs to run on each node. However, the placement of this service is not strict.

The global deployment mode is useful for this purpose. An instance of the service will run on every node, but it is not important to lock a service to a specific node. The same applies to the other stream operator services. Scripts B.17 to B.18 on the following page are deployed as global services for the *distanceTrigger* and *deadNodeDetector* stream operators specifically.

```
89  nats-processor-position-update-3:
90    image: "pietersynthesis/nats-swarm-positionupdate:armv7l"
91    networks:
92      - main
93    depends_on:
94      - nats-streaming-1
95    deploy:
96      mode: replicated
97      replicas: 1
98      placement:
99        constraints:
100      - node.hostname == pi-3
```

Script B.15 Application stack definition in docker-compose YAML: 8 of 11.

```
101  nats-processor-distance-calc:
102    image: "pietersynthesis/nats-swarm-distancecalculator:armv7l"
103    networks:
104      - main
105    depends_on:
106      - nats-streaming-1
107    deploy:
108      mode: global
```

Script B.16 Application stack definition in docker-compose YAML: 9 of 11.

```
109  nats-processor-distance-trigger:
110    image: "pietersynthesis/nats-swarm-distancetrigger:armv7l"
111    networks:
112      - main
113    depends_on:
114      - nats-streaming-1
115    deploy:
116      mode: global
```

Script B.17 Application stack definition in docker-compose YAML: 10 of 11.

```
117  nats-processor-deadnode-detector:
118    image: "pietersynthesis/nats-swarm-deadnodedetector:armv7l"
119    networks:
120      - main
121    depends_on:
122      - nats-streaming-1
123    deploy:
124      mode: global
```

Script B.18 Application stack definition in docker-compose YAML: 11 of 11.