RIGA TECHNICAL UNIVERSITY

Dmitrijs Rjazanovs

ANALYSIS OF FAULT-TOLERANT ALGORITHMS FOR THE DEVELOPMENT OF COOPERATIVE MOBILE NETWORKS

Doctoral Thesis

RTU Press
Riga 2025

RIGA TECHNICAL UNIVERSITY
Faculty of Computer Science, Information Technology and Energy
Institute of Photonics, Electronics and Telecommunication

Dmitrijs Rjazanovs
Doctoral Student of the Study Programme "Telecommunications"

ANALYSIS OF FAULT-TOLERANT ALGORITHMS FOR THE DEVELOPMENT OF COOPERATIVE MOBILE NETWORKS

Doctoral Thesis

Scientific supervisors
Associate Professor Dr. sc. ing.
ALEKSANDRS IPATOVS
Professor Dr. habil. sc. ing.
ERNESTS PĒTERSONS

RTU Press
Riga 2025

DOCTORAL THESIS PROPOSED TO RIGA TECHNICAL UNIVERSITY FOR THE PROMOTION TO THE SCIENTIFIC DEGREE OF DOCTOR OF SCIENCE

To be granted the scientific degree of Doctor of Science (Ph. D.), the present Doctoral Thesis has been submitted for defence at the open meeting of RTU Promotion Council on 28 November 2025 at 11:00 AM at the Faculty of Computer Science, Information Technology and Energy (FCSITE) of Riga Technical University (RTU), 12 Azenes Str., Room 201.

OFFICIAL REVIEWERS

Professor Dr.sc.ing. Vjačeslavs Bobrovs
Riga Technical University, Latvia

Professor Dr.sc.ing. Toledo Moreo Rafael
Polytechnic University of Cartagena (UPCT), Spain

Professor. Dr. ing. habil Mehmet Ercan Altinsoy
Dresden University of Technology (TUD), Germany

DECLARATION OF ACADEMIC INTEGRITY

I hereby declare that the Doctoral Thesis submitted for review to Riga Technical University for promotion to the scientific degree of Doctor of Science (Ph. D.) is my own. I confirm that this Doctoral Thesis has not been submitted to any other university for promotion to a scientific degree.

Dmitrijs Rjazanovs ……. (signature)
Date: ……

The Doctoral Thesis has been written in English. It consists of an introduction, 5 chapters, conclusion, 33 figures, 12 tables, and 5 appendices; the total number of pages is 101, not including appendices. The Bibliography contains 61 titles.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to the supervisors of my work, Dr. habil. sc. ing., professor Ernests Pētersons and Dr. sc. Ing., Aleksandrs Ipatovs. Without them, my beginnings might never have become something greater. I also extend my sincere respect to my colleagues from the Institute of Telecommunications, as well as to all the staff of RTU.

Finally, I am endlessly grateful to my wife, my beloved children, and my parents. Thank you for allowing me to complete this work!

PATEICĪBA

Izsaku sirsnīgu pateicību sava darba vadītājiem, Dr. habil. sc. ing., profesoram Ernestam Pētersonam un Dr. sc. ing., Aleksandram Ipatovam. Bez viņiem mani sākumi, iespējams, nekad nebūtu kļuvuši par kaut ko lielāku. Tāpat izsaku patiesu cieņu kolēģiem no Telekomunikāciju institūta, kā arī visam RTU personālam.

Visbeidzot, esmu bezgalīgi pateicīgs savai sievai, maniem mīļajiem bērniem un vecākiem. Paldies, ka ļāvāt man pabeigt šo darbu!

ANNOTATION

Title of the thesis:

"Analysis of fault-tolerant algorithms for the development of cooperative mobile networks"

Author:

Dmitrijs Rjazanovs

Contents of the work:

This Thesis addresses the complexity of engineering approaches for fault-tolerant mobile networks. The problem originates from conventional IT systems, where, despite the maturity of software development practices, fault tolerance often does not receive adequate attention. This perspective is extended even further to the field of cooperative mobile network solutions. The primary goal of this research is to advance efficient, fault-tolerant solutions for cooperative mobile networks and provide insights into the efficiency of fault-tolerant consensus, exploring its various forms and practical implementation strategies.

The research specifically focuses on measuring the efficiency of eventual and synchronous leader election algorithms in cooperative UAV patrol missions. Insights are provided into the characteristics of the eventual algorithm's convergence process, the effects of different timeout strategies on mission quality, the dynamic characteristics of cooperative UAV networks, and the impact of AI accuracy and performance on mission outcomes. Additionally, the consensus problem is analyzed in the context of smart vehicle cooperative crash avoidance scenarios, exploring whether the Byzantine error model is necessary in V2V communications. A method for modeling and measuring fail-stop flooding consensus performance in this scenario is developed, and the results are analyzed. To achieve these objectives, a novel, real-time, Docker-based simulation was designed and developed. A queuing network model was created to analyze the dynamic characteristics of the cooperative UAV solution, while a custom NS3 simulation was developed to evaluate consensus in V2V scenarios.

ANOTĀCIJA

Darba nosaukums:

"Defekttolerantu algoritmu analīze kooperatīvo mobilo tīklu izveidei"

Darba autors:

Dmitrijs Rjazanovs

Darba saturs:

Promocijas darbs pievēršas inženiertehnisko pieeju sarežģītībai defekttolerantu mobilo tīklu izstrādē. Problēma sākas ar tradicionālajām IT sistēmām, kurās, neskatoties uz programmatūras izstrādes prakses attīstības līmeni, defekttolerancei bieži netiek veltīta pietiekama uzmanība. Šī perspektīva tiek paplašināta līdz kooperatīvo mobilo tīklu risinājumu jomai. Pētījuma galvenais mērķis ir attīstīt efektīvus, defekttolerantus risinājumus kooperatīviem mobilajiem tīkliem un sniegt ieskatu defekttoleranta konsensa efektivitātē, izpētot dažādas tā formas un praktiskās ieviešanas stratēģijas.

Pētījums īpaši pievēršas izrietoši un sinhrono līdera izvēles algoritmu efektivitātes novērtēšanai kooperatīvajās bezpilota lidaparātu (BPL) patrulēšanas misijās. Tiek sniegti ieskati par izrietošo algoritma konverģences procesa raksturlielumiem, dažādu pauzu stratēģiju ietekmi uz misijas kvalitāti, kooperatīvo BPL tīklu dinamiskajām īpašībām un MI precizitātes un veiktspējas ietekmi uz misijas rezultātiem. Papildus tam konsensa problēma tiek analizēta viedo transportlīdzekļu kooperatīvas sadursmes novēršanas scenāriju kontekstā, pētot, vai Bizantijas kļūdu modelis ir nepieciešams *V2V* sakaros. Tiek izstrādāta metode, lai modelētu un mērītu plūdu konsensa veiktspēju šajā scenārijā, un rezultāti tiek analizēti. Lai sasniegtu šos mērķus, tika sagatavota un izstrādāta jauna reāllaika *Docker* balstīta simulācija. Tika izveidots rindas tīkla modelis, lai analizētu kooperatīvā BPL risinājuma dinamiskās īpašības, savukārt pielāgota *NS3* simulācija tika izstrādāta, lai novērtētu konsensu *V2V* scenārijos.

CONTENTS

ABBREVIATIONS

A
ACK – Acknowledgement
AI – Artificial Intelligence
API – Application Programming Interface
ASCII – American Standard Code for Information Interchange

B
BEB – Best Effort Broadcast
BFT – Byzantine Fault Tolerance
BOLD – Bio-inspired Optimized Leader Election for Multiple Drones

C
CAP – Consistency, Availability, Partition Tolerance
C-ITS – Cooperative Intelligent Transport Systems
CNN – Convolutional Neural Network
CPU – Central Processing Unit
CDF – Cumulative Distribution Function

D
DB – Database
DSRC – Dedicated Short-Range Communications
DT – Delta Timeout

E
EU – European Union
ETSI – European Telecommunications Standards Institute

F
FOV – Field of View

G
GDPR – General Data Protection Regulation
GNQN – Gordon-Newell Queueing Network
GPS – Global Positioning System

H
HIL – Hardware in the Loop
HWMP – Hybrid Wireless Mesh Protocol

I
IBM – International Business Machines
ICMP – Internet Control Message Protocol
ICE – Interactive Connection Establishment
ICO – Initial Coin Offering
ID – Identifier
IEEE – Institute of Electrical and Electronics Engineers
IOT – Internet of Things
IP – Internet Protocol
IS – Infinite Server

K
KPI – Key Performance Indicator

L
LMT – Latvian Mobile Telephone

M
MANET – Mobile Ad Hoc Network
MVA – Mean Value Analysis

N
NAT – Network Address Translation
NS-3 – Network Simulator 3

O
OS – Operating System

P
PDF – Probability Density Function
PHY – Physical Layer
PKI – Public Key Infrastructure
PMF – Probability Mass Function
p2p – Peer-to-Peer

Q
QN – Queueing Network

R
RAM – Random Access Memory
ROS – Robot Operating System
RSU – Roadside Unit

S
SIM – Subscriber Identity Module
SSD – Solid State Drive
STUN – Session Traversal Utilities for NAT
SV – Smart Vehicles

T
TCP – Transmission Control Protocol
TURN – Traversal Using Relays around NAT

U
UDP – User Datagram Protocol
UAV – Unmanned Aerial Vehicles
UpnP – Universal Plug and Play
UTF – Unicode Transformation Format

V
V2V – Vehicle-to-Vehicle Communication
V2X – Vehicle-to-Everything
VANET – Vehicular Ad Hoc Network

W
WAN – Wide Area Network

4G – Fourth Generation Cellular Technology
5G – Fifth Generation Cellular Technology
4K – Ultra High-Definition Video
2D – Two-Dimensional
3D – Three-Dimensional

1. INTRODUCTION

This thesis explores the challenge of identifying and implementing sufficiently robust fault-tolerant algorithms in the context of cooperative mobile networks. Both fault-tolerant algorithms and cooperative mobile networks encompass a vast range of use cases; therefore, this research focuses on selected scenarios with the aim of deriving general insights applicable to other contexts as well. The thesis presents both theoretical and practical contributions, with a primary emphasis on practical implementation. The first scenario examines cooperative UAV perimeter patrol, including simulation and a leader election algorithm. The second scenario addresses the problem of achieving consensus during emergency situations in smart vehicular networks. The goal of this chapter is to briefly introduce the research problem, provide contextual background, and explain its significance.

Fault tolerance is the capability of a system to maintain proper operation despite the occurrence of failures or faults in one or more of its components. Common examples of fault-tolerant algorithms include reliable broadcast, consensus, leader election, and other mechanisms rooted in distributed systems theory.

Traditionally, fault-tolerant algorithms are discussed in the context of distributed systems, where multiple components operate independently but are interconnected via a network, forming part of a larger system. A system may evolve from a single-server to a multi-server architecture for two primary reasons:

- Artificial Redundancy: Engineers intentionally add redundant resources to enhance fault tolerance and prevent failures. For example, in civil engineering, large ships are designed with bulkheads, ensuring that if a ship collides with an iceberg, only a portion is damaged, allowing the ship to continue operating. Redundancy can be applied in two dimensions:
    - Spatial Redundancy: Multiple components operate simultaneously (e.g., multiple servers).
    - Temporal Redundancy: Actions or computations are repeated over time (e.g., retries in communication).
- Natural Distribution: Some applications are inherently distributed, such as collaborative document editing, where multiple users work on a document from different locations.

Fault tolerance can also be applied to single-computer systems through mechanisms such as exception handling, process isolation, watchdog timers, and more. However, this thesis focuses specifically on fault tolerance in distributed systems.

Reliable broadcast is a distributed system primitive that can be used to build other high-level primitives, such as consensus. Its primary objective is to ensure that all recipients receive the original message. There are many variations of this algorithm, including fail-silent basic broadcast, fail-stop lazy reliable broadcast, fail-silent eager reliable broadcast, uniform reliable broadcast, fail-recovery reliable broadcast, probabilistic broadcast, causal broadcast, Byzantine consistent broadcast, and total order broadcast [1]. The differences between these forms will be

discussed in detail later, but similar variations also apply to other fault-tolerant algorithms. Each type of algorithm has its own assumptions and guarantees. Traditionally, all assumptions and guarantees in distributed systems are classified as liveness and safety [2]. This classification also intersects with multi-threaded programming, where synchronization primitives maintain similar classes of guarantees [3].

Unlike broadcast, which ensures message delivery, consensus requires nodes to actively agree on a single value despite uncertainty and partial failures. This makes consensus inherently harder, especially in asynchronous systems where the famous FLP impossibility result proves that deterministic consensus cannot be achieved with even one faulty process [4]. As a result, consensus algorithms often involve randomized techniques, leader election, quorum-based voting, or timeout-driven progress mechanisms. In mobile networks, consensus faces additional challenges due to dynamic topology and variable connectivity, making traditional solutions like Paxos or Raft difficult to apply directly [5]. This motivates the exploration of lightweight, adaptive consensus mechanisms that can tolerate delays, message loss, or temporarily disconnected nodes while still maintaining consistency.

Traditionally, the application of fault-tolerant algorithms was discussed and developed in the context of database clusters. There are limits to how far a single database server can be scaled vertically. At some point, multiple servers may be needed to share the load of requests on the database. This introduces the problem of database state replication, which is a form of consensus with numerous variations and implementations [6]. Since the 1970s, many algorithms and practical systems have been developed to address this problem. The theory and practice behind these systems are quite mature; nevertheless, research in this area continues [7].

Distributed systems are a fundamental component of modern technology, supporting a wide range of applications such as cloud computing, autonomous vehicles, PageRank, smart power grids, state estimation, UAV coordination, multi-agent control, load balancing, and blockchain. Originating from early work on database replication, distributed systems have evolved to support increasingly complex and decentralized architectures. This evolution has significantly influenced both technological development and societal interaction with digital systems. While they enable many of today's autonomous and distributed technologies, distributed systems also continue to present both theoretical and practical challenges [8]. This thesis aims to examine how core concepts—such as consensus—can vary in implementation and behavior depending on the specific application context, whether in vehicular networks, UAV operations, or permissionless peer-to-peer systems.

Blockchain represents not merely another technology but a novel theoretical direction in distributed systems, offering a probabilistic solution to the same consensus primitive in permissionless peer-to-peer networks, as first introduced by [9]. Despite negative associations with blockchain, such as its use in gambling and ICO-related fraud, it presents significant challenges and opportunities for scientists and engineers worldwide. Beyond the classic application of blockchain in financial transactions, compelling use cases have emerged in IoT and smart contracts, such as [10] and [11]. Governments and industries are increasingly investing in blockchain and IoT integration to enhance security, transparency, and efficiency [12]. Initiatives like the EU's GDPR compliance for IoT and private blockchain projects, such

as IBM's Food Trust and Maersk's supply chain pilots, exemplify this progress [13] [14]. The automotive and logistics sectors, in particular, are driving adoption with applications in vehicle communication and supply chain tracking [12].

Cooperative mobile networks consist of autonomous, mobile agents—such as vehicles, drones, or robots—that interact and coordinate to achieve shared objectives, possibly, without centralized control. These networks operate in dynamic environments with changing topologies, unreliable communication links, and limited resources. To function effectively, they require decentralized algorithms capable of handling failures, making decisions collaboratively, and adapting to real-time conditions, creating a unique intersection between mobility, autonomy, and distributed fault-tolerant computing.

Autonomous smart vehicles are one of big challenges for both industry and academia. The operation of autonomous vehicles already relies on distributed consensus mechanisms for critical tasks such as platooning, collision avoidance, and cooperative decision-making, highlighting their role as components of a collaborative network rather than isolated entities [15]. These mechanisms ensure real-time coordination, traffic efficiency, and safety, yet face significant challenges. Privacy concerns arise from the extensive data sharing required between vehicles, while performance demands necessitate robust, low-latency processing for dynamic conditions. Furthermore, reliability is crucial to prevent system failures that could lead to catastrophic outcomes. Overcoming these challenges is essential to advance the implementation of secure and scalable distributed systems in interconnected SV ecosystems.

For the past ten years, personal electric UAVs have gained widespread popularity globally. The primary catalyst for this surge is attributed to the advancements in the lithium polymer battery industry [16] [17]. Initially, UAVs could only sustain flights for a few minutes; however, the current standard exceeds 20 minutes [18]. Cooperative UAV systems rely heavily on distributed consensus mechanisms to enable critical operations such as formation flying, coordinated search and rescue missions, and dynamic task allocation [19] [20]. These mechanisms ensure real-time coordination, collision avoidance, and efficient resource utilization across a network of UAVs operating in dynamic and unpredictable environments.

Despite the critical role that fault-tolerant primitives play in modern IT and cyber-physical systems, there remains a notable gap in the practical understanding of their theory and application. Even in typical enterprise systems—which are increasingly distributed—basic design mistakes are often made, leading to poor fault tolerance, reduced system reliability, and higher development and maintenance costs [21]. There are several reasons for this. One is that these primitives are usually abstracted away by frameworks and libraries, allowing engineers to use them without understanding the underlying assumptions and limitations. However, in practice, such abstractions rarely provide complete, end-to-end fault tolerance. A common example is the misuse of message brokers. Engineers may assume that posting a message to a broker after a transaction is sufficient, overlooking the fact that network failures can prevent message delivery. Without additional mechanisms, such as the outbox pattern [22], data consistency and message reliability cannot be guaranteed. Another contributing factor is that systems are often designed with a focus on the "happy path", while failure scenarios are

postponed or overlooked, especially in later development phases when budget and time are limited.

In the following subchapter, all necessary theoretical foundations required to understand the subsequent chapters and the researched scenarios will be discussed. Next, existing work on this topic will be reviewed. Finally, the last subchapter will present a summary of the achieved results and outline the overall structure of the thesis.

1.1. Fault tolerant algorithms

FAULTLESS SYSTEMS DOES NOT EXISTS

Computers and software systems possess an immense state space, making it nearly impossible to predict and handle all possible scenarios gracefully. This complexity is one of the primary reasons for the existence of the "hacking" industry. As Cheswick and Bellovin noted in 1994, "Any program, no matter how innocuous it seems, can harbor security holes" [23]. Engineers naturally focus on the primary use case, addressing the goals and core functionalities that the system must achieve. This task is often so complex that it consumes all the available attention and energy. Once the primary functionality is implemented, there is frequently little time or inclination left to consider negative scenarios and corner cases. Despite the extensive theoretical knowledge about faults, this tendency to overlook negative scenarios is highlighted by the continued relevance of "The Fallacies of Distributed Computing" [24]. These fallacies demonstrate the persistent underestimation of failure modes in distributed systems.

Another revealing aspect of distributed systems is the contrast between early theoretical breakthroughs and the delayed understanding of high-level principles. Remarkable discoveries, such as solutions to the Byzantine Generals Problem [25], the FLP impossibility result [4], and Lamport's logical clocks [26], were made before 1985. However, the CAP theorem—arguably the most widely applied and general discovery in distributed systems—was not published until 2000. This delay underscores the fact that while researchers were uncovering deep algorithmic principles, it took much longer to comprehend and formalize overarching, high-level laws that guide the design and behavior of distributed systems.

FAULT CLASSES AND HIERARCHY

We can see on figure 1.1 the classification and hierarchy of the possible system faults. **Crash Faults** are the simplest type of failure which occurs when a component, such as a server or process, ceases to function. A system experiencing a crash fault stops executing operations but does not perform any incorrect or unintended actions.

Omission Faults: These faults extend beyond crash faults and include scenarios where a system component fails to send or receive messages. Omission faults disrupt communication between processes and can arise from network failures or process delays.

Crash with Recovery: In this category, a system that experiences a crash fault eventually recovers and resumes operation. While recovery improves fault handling, it introduces complexity in ensuring consistency and synchronization among system components.

Eavesdropping: This fault type involves an adversary intercepting and observing communication between components without altering the data. Although not disruptive, eavesdropping raises security and privacy concerns, particularly in sensitive applications.

Arbitrary Faults (Byzantine Faults): The most challenging class of faults encompasses arbitrary or Byzantine faults. In these cases, a faulty component can behave unpredictably, including sending conflicting or maliciously crafted messages. Managing arbitrary faults requires sophisticated algorithms and mechanisms, such as Byzantine fault tolerance, to maintain the integrity and consistency of the system.

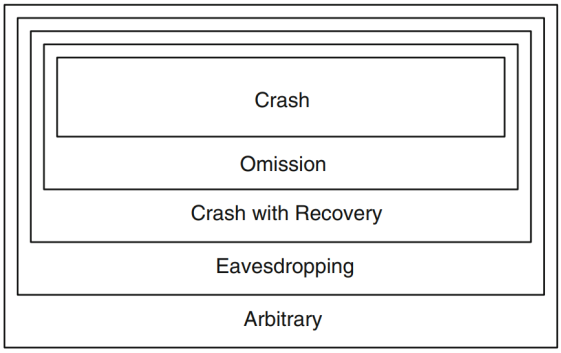

fig. 1.1. Fault classes in distributed system theory [1].

TWO GENERALS' PROBLEM

One of the famous problems in computer science is the "Two Generals' Problem" [27]. This problem serves as both a model for understanding the limits of what can be achieved algorithmically in distributed systems and a proof of impossibility. It is essential for understanding the concept of eventual consistency (which will be discussed in the next chapters) and knowing when it is appropriate to use it. The Two Generals Problem shows that two parties communicating over an unreliable channel cannot guarantee agreement on a coordinated action (e.g., attack time), because any message or acknowledgment might be lost, making reliable confirmation impossible. Informal proof can be following. Assume there exists an algorithm that guarantees consensus between two parties using at most N messages over an unreliable network. Since the algorithm terminates after N messages, there must be a last message that ensures both parties reach agreement. But if any message can be lost, then this final message might be lost too. In that case, the algorithm would have to work correctly even without that last message, meaning it could have achieved consensus in $N-1$ messages — a contradiction. Thus, no such algorithm can exist.

In practical applications, the network is often assumed to be reliable. More precisely, reliability is guaranteed in the eventual delivery sense: every message sent will eventually be delivered, though delays or reordering may occur.

UNRELIABLE NETWORK AND TCP

In practical distributed systems, communication is typically built upon protocols like TCP, which provide a high level of reliability through retransmission, ordering, and error detection mechanisms. TCP is often treated as a "reliable" channel in engineering terms, assuming that any message sent will eventually be delivered unless the connection is explicitly closed. This eventual delivery guarantee allows developers to build systems that tolerate transient network issues while maintaining functional application-level semantics.

However, such protocols do not eliminate the fundamental uncertainty in achieving agreement across nodes, as highlighted in the previous chapter. TCP ensures message delivery under typical network conditions, but it does not guarantee mutual knowledge of delivery. For example, during TCP's three-way handshake, the loss of the final acknowledgment may leave one party believing the connection is established, while the other remains unaware. This scenario illustrates that TCP, although reliable in practice, cannot provide the certainty required for consensus in the formal sense. It relies on timeouts and retries, and may silently fail in edge cases, leading to asymmetric views of the system state.

This distinction has important implications for the design of fault-tolerant distributed algorithms. Systems built on top of TCP must not assume perfect agreement based solely on successful connections or acknowledgments. Instead, they must incorporate additional mechanisms—such as failure detectors, quorum-based voting, or consensus protocols like Paxos or Raft—to ensure correctness under uncertainty. Recognizing the limits of practical reliability is crucial when translating theoretical models into real-world deployments.

ALGORITHM CLASSES

While discussing algorithms it is convenient to use classification that can quickly tell something about the assumptions made by this algorithm. [1] provides one such possible classification:

1. **fail-stop** algorithms, designed under the assumption that processes can fail by crashing but the crashes can be reliably detected by all the other processes;
2. **fail-silent** algorithms, where process crashes can never be reliably detected;
3. **fail-noisy** algorithms, where processes can fail by crashing and the crashes can be detected, but not always in an accurate manner (accuracy is only eventual);
4. **fail-recovery** algorithms, where processes can crash and later recover and still participate in the algorithm;
5. **fail-arbitrary** algorithms, where processes can deviate arbitrarily from the protocol specification and act in malicious, adversarial ways;
6. **randomized** algorithms, where in addition to the classes presented so far, processes may make probabilistic choices by using a source of randomness.

1.2. Existing work

In this chapter, we are going to present the existing theory on leader election in UAV groups as well as consensus problem in emergency V2V scenarios.

LEADER ELECTION

Leader election is a foundational problem in distributed systems, concerned with selecting one distinguished process (or node) among a set of distributed and potentially identical processes [2]. This process, once elected, assumes the role of coordinator or leader and is responsible for performing specific centralized tasks such as synchronization, decision-making, or initiating coordinated actions.

The problem arises in settings where multiple processes operate concurrently and independently without global knowledge of the system. To be effective, leader election protocols must ensure that all correct processes eventually agree on the same leader, and that this leader is unique and active. The classical formulation of the problem includes three desirable properties: **uniqueness** (only one leader at a time), **agreement** (all correct processes agree on the leader), and **termination** (a leader is eventually elected) [3].

Leader election algorithms are often classified based on timing assumptions and failure conditions. In synchronous systems, where message delivery and processing delays are bounded, deterministic leader election is possible. However, in asynchronous systems, where no such bounds exist, it is impossible to guarantee both safety and liveness in the presence of even a single crash failure, as formalized in the Fischer-Lynch-Paterson (FLP) impossibility result [3].

To make leader election feasible in partially synchronous systems [28], researchers introduce failure detectors, which act as abstract modules providing (possibly unreliable) information about process failures. Two well-known classes of failure detectors are the Perfect Failure Detector (P) and the Eventually Perfect Failure Detector ($\Diamond P$) [1]:

- A Perfect Failure Detector (P) satisfies:
    - Strong completeness: Eventually, every process that crashes is permanently detected by every correct process.
    - Strong accuracy: If a process is detected as crashed by any process, then it has indeed crashed.
- An Eventually Perfect Failure Detector ($\Diamond P$) satisfies:
    - Strong completeness: Eventually, every process that crashes is permanently suspected by every correct process.
    - Eventual strong accuracy: Eventually, no correct process is suspected by any correct process.

Failure detectors are instrumental in enabling leader election. The basic requirement of a leader election module is to eventually elect a correct leader and to avoid electing multiple leaders simultaneously. Formally, a leader election module must satisfy:

- **Eventual detection:** Either there is no correct process, or eventually some correct process is elected as the leader.
- **Accuracy:** If a process is elected leader, then all previously elected leaders have crashed.

A particularly useful abstraction in asynchronous systems is the **eventual leader detector Ω**, which satisfies:

- Eventual accuracy: There exists a time after which every correct process trusts some correct process as the leader.
- Eventual agreement: There exists a time after which no two correct processes trust different correct processes as leader.

These formal properties provide the basis for implementing leader election protocols that remain correct despite network unreliability and process failures.

In the context of UAV networks, while many real-world systems exhibit mobility and dynamic topologies, this study deliberately focuses on a static connected network topology, where only communication delays and occasional crashes affect system behavior. This controlled setup allows for a precise evaluation of how failure detector accuracy and timeliness influence mission-level coordination.

We investigate two leader election strategies:

1. A leader election algorithm based on a **Perfect Failure Detector (*P*)**, which ensures accurate detection of process crashes.
2. A leader election algorithm based on an **Eventually Perfect Failure Detector (◇P)**, which may initially suspect correct processes but stabilize over time.

Although both strategies guarantee convergence to a single correct leader, their transient behaviors differ significantly under conditions of network delay. The core research question addressed in this study is: how do these differences impact mission-level performance metrics in cooperative UAV operations?

Instead of evaluating these protocols only in terms of convergence time or message complexity, this work assesses their impact on mission KPIs such as threat registration, monitoring time, and communication cost. This approach bridges theoretical correctness with practical effectiveness, offering insight into how failure detection properties translate into real-world UAV coordination outcomes.

LEADER ELECTION IN UAV NETWORKS

As seen in the previous section, all leader election algorithms can be split into two main categories: those that assume a synchronous network and those that operate in a partially synchronous network. Each category contains various implementations. Leader election has its roots in network components, such as IBM's token ring [29], where computers were connected in a physical ring or star topology. In the token ring, leader election was used to manage token circulation, ensuring that one node controlled access to the shared communication medium. Another example is leader election in a database cluster. A classic example here is the Bully algorithm [30], which assumes a general topology and a synchronous network, relying on static timeouts to detect a dead server. Another example in database clusters is the Raft algorithm [5], where a leader is elected to ensure efficient state machine replication (total order broadcast). Unlike the Bully algorithm, Raft assumes a partially synchronous network, but it requires that more than half of the nodes be online for the algorithm to function correctly.

In [31] author proposed a Raft based leader election and clustering scheme that is resilient to UAV failures. Although their approach theoretically supports convergence and fault

tolerance, it remains abstract, omitting modeling of message delays, false suspicions, or the cost of reconfiguration under partial asynchrony. Similarly, authors of [32] present a simulation-based study of a voting-based leader election scheme for UAV swarms in a lead-follow configuration under constrained communication. Using an agent-based model in Pygame, it simulates swarms of 30 to 100 UAVs, with leader election triggered upon leader failure. Followers calculate qualifications and vote to elect a new leader. The study examines the impact of communication range on performance metrics like completion rate, success rate, average rank, and simulation time, comparing the proposed method with a Raft-based method. Results show the voting-based approach is faster and more successful, even with reduced communication requirements.

Mousavi et al. [33] addressed coalition formation in UAV networks using a multi-objective optimization algorithm based on quantum evolutionary techniques. Their work focused on optimizing cost, trust, and reliability when assigning UAVs to task coalitions. While the model supported large-scale task assignment, it did not explicitly model leader election or fault tolerance, nor did it incorporate network delay or its impact on decision propagation. Ganesan et al. [34] proposed the BOLD algorithm, which combines particle swarm optimization and spider monkey heuristics to elect energy-efficient cluster heads in dynamic UAV networks. Although the algorithm improves energy balance and extends operational lifespan, it is not designed for fault resilience or transient consistency. The absence of failure detection mechanisms or recovery logic restricts its applicability in delay-prone or failure-prone mission environments. Wang et al. [35] introduced a secure UAV-aided cluster head election mechanism aimed at protecting wireless sensor networks from compromised nodes. Their framework assumes a UAV that functions as an external authority with direct access to node health and energy metrics. While valuable in constrained IoT contexts, this centralized model is not applicable in autonomous UAV swarms that lack supervisory infrastructure and must elect leaders collaboratively.

In general, prior work on leader election in UAV systems can be divided into three broad categories: (1) optimization-based leader selection focused on energy or resource efficiency; (2) secure or supervised schemes that assume centralized control or perfect observation; and (3) consensus-based models that describe correctness but are evaluated only under ideal conditions. Across all these categories, a common limitation is the lack of empirical evaluation under partial synchrony, particularly with regard to the type of failure detectors employed and their practical effect on mission-critical metrics.

This study addresses the gap by comparing two formally sound leader election strategies derived from distinct failure detection assumptions: one assuming a Perfect Failure Detector (P), and the other based on an Eventually Perfect Failure Detector ($\Diamond P$). These strategies are implemented within a controlled UAV simulation environment where the network is static but subject to message delays and occasional crashes. The study examines how the transient behaviors of these algorithms influence key mission-level indicators such as threat detection continuity, leader stability, and communication cost.

By aligning theoretical models of failure detection with empirical performance data, this work contributes a unique and needed perspective: it shows how the properties of failure

detectors translate into operational effectiveness, thus bridging the gap between distributed computing theory and real-world cooperative mobile networks.

CONSENSUS

Consensus is a fundamental problem in distributed computing, where a group of processes must agree on a single value despite failures (fig. 1.2) and communication delays. Formally, a consensus algorithm must satisfy the following properties [3]:

- **Agreement**: No two correct processes decide differently.
- **Validity**: If all correct processes propose the same value, it must be the decided value.
- **Termination**: Every correct process eventually decides.

The solvability of consensus depends on the assumptions made about failures and synchrony. Three failure models are commonly discussed [1]:

1. **Byzantine model**: Processes may behave arbitrarily or maliciously.
2. **Crash-stop model**: Processes may halt (crash) and never recover.
3. **Crash-recovery model**: Processes may crash and later recover, possibly with memory loss.

For vehicular systems, the crash-stop model is the most realistic model. It reflects communication disruptions or software faults that prevent a node from continuing protocol execution. Unlike Byzantine faults, crash-stop failures are simpler to detect and model.

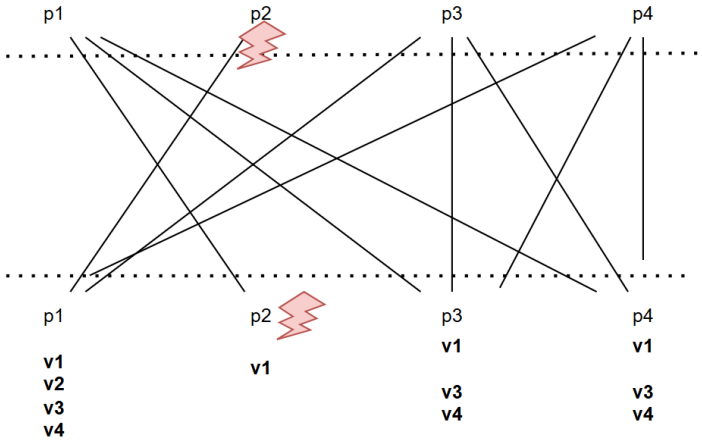

fig. 1.2. Example of nodes not reaching agreement during broadcast due to crash.

In terms of timing, consensus is impossible in purely asynchronous systems with even one faulty process, as shown by the FLP impossibility result [4]. To overcome this, most practical consensus protocols assume **partial synchrony**—the system may behave asynchronously but eventually becomes synchronous. This assumption, together with crash-stop faults, allows the design of robust consensus protocols with bounded delay.

Consensus algorithms can be classified by:

- **Failure model**: Byzantine vs crash-stop

- **Synchrony**: synchronous, partially synchronous
- **Communication strategy**: point-to-point vs broadcast, single leader vs symmetric roles
- **Message and time complexity.**

In vehicular networks, especially V2V scenarios, consensus is used for safety-critical coordination such as collision avoidance or cooperative braking. These scenarios impose strict real-time requirements, typically under 500 ms, and must tolerate communication delays and unreliable links [15]. This study focuses on crash-stop consensus in synchronous systems, where message delays are bounded and failures are reliably detected, as required by the Flooding Consensus algorithm [1]. The performance of this protocol under varying topologies, node counts, and retry configurations is evaluated to assess its feasibility in emergency vehicular scenarios.

CONSENSUS AND VEHICULAR NETWORKS

The need for rapid, fault-tolerant agreement in vehicular networks has led to increased interest in lightweight consensus protocols suitable for highly dynamic wireless environments. Most existing work assumes a crash-stop failure model and builds on partially synchronous or synchronous communication settings. Byzantine-resilient consensus is rarely addressed in this context, largely due to its high communication overhead and the assumption that V2V failures are typically benign (e.g., due to interference or range loss).

This paper [20] proposes an adaptive Raft protocol for wireless networks. The algorithm assumes a partially synchronous network with variable delays, as it accounts for dynamic network conditions and varying transmission times. The network is multi-hop, as the adaptive Raft protocol uses relay nodes and routing paths for state synchronization and communication between nodes. Using matlab simulation, authors measure full consensus reliability, log replication success rate, and latency of the adaptive Raft protocol in the wireless network and compare it with original Raft algorithm. While the work is fundamental and comprehensive, the authors do not specifically address possibility of one-time consensus without leader election that might better fit for V2V scenario. Raft is mainly designed for log replication which might be unnecessary for V2V scenario, because vehicles only need latest data and do not need to maintain large database with incoming operations. Other byzantine fault tolerant approaches will be described in the chapter 5.2.

1.3. Summary and structure of the thesis

Considering the above-mentioned facts **the objective of the doctoral thesis** was set:

Research existing consensus algorithms, assess their applicability in advanced cooperative mobile network scenarios, and develop simulations to evaluate their performance under harsh conditions for potential development in real deployments.

To accomplish the goal, the following main tasks were defined:

1. Conduct an in-depth study of distributed systems' fundamental principles, including fault-tolerant algorithms, network models, and boundary results in consensus theory.
2. Investigate the latest advancements in distributed systems, including blockchain-based consensus algorithms, and evaluate their applicability in mobile network environments.
3. Research and apply queueing theory as a mathematical foundation for evaluating network performance and algorithm efficiency in mobile networks.
4. Explore the theoretical foundations and practical implementations of UAV-based mobile networks, including mission planning, communication, and cooperative scenarios.
5. Investigate V2X communication, covering both theoretical foundations and practical technologies used in smart vehicular networks.
6. Evaluate existing simulation frameworks and methods for modeling cooperative UAV missions, focusing on flexibility, scalability, and network reliability.
7. Design and develop a simulation environment for cooperative UAV missions, capable of testing various fault conditions, including UAV crashes, unstable networks, and delayed communication.
8. Develop a methodology to compare synchronous and partially synchronous consensus algorithms in the context of cooperative UAV perimeter patrol.
9. Design and implement a simulation model for consensus in vehicular networks (V2V), focusing on fault-tolerant agreement under crash-stop conditions.
10. Conduct extensive experiments using the developed simulations, analyze the results, and derive conclusions regarding the practical applicability of consensus algorithms in real-world deployments.

RESEARCH METHODOLOGY

During the completion of the defined tasks, theoretical correctness, performance, assumptions, and guarantees of distributed algorithms were thoroughly analyzed. This analysis was combined with software design principles to develop simulations and create various scenarios for quantitatively comparing the effectiveness of distributed systems. Mathematical calculations were conducted to support the simulation results, and field experiments were carried out to validate network behavior under real-world conditions.

To evaluate NAT traversal in the field, the following technologies were employed: the C# programming language, Azure cloud virtual machine services, an Ubuntu-based laptop, a Windows-based laptop, the UDP network stack, and SIM cards from multiple mobile operators. This setup enabled comprehensive testing of NAT traversal and peer-to-peer connectivity under different network conditions.

For the implementation of the cooperative UAV patrol mission simulation, Docker was utilized for containerized deployment, while the simulation logic was developed using C# and .NET 8, with the XUnit testing framework ensuring reliability and repeatability of results. The simulation modeled network behavior using a log-normal probability distribution to represent

transmission delays, and a sample size calculation formula was used to determine the minimum number of simulation runs required for statistically significant results. Core cooperation mechanisms were built on the basis of eventual and perfect failure detectors, as well as eventual and perfect leader detectors, which provided a robust foundation for UAV coordination.

Python, in combination with the Storybook and VS Code environment, was used for processing simulation results and calculating the dynamic characteristics of the UAV network. The mathematical foundation was based on a Closed Gordon-Newell Queueing Network (QN) model, which defined the UAV network's key components. Buzen's convolution algorithm was employed to calculate throughput, utilization, and other essential properties of the QN. Furthermore, a custom Python script was developed to calculate the probability distribution of QN response times using the QN normalization constant.

The NS-3 framework was applied to simulate all-to-all custom UDP and HWMP-based broadcast algorithms within an 802.11s mesh network. Different connectivity graph diameters and topologies were tested to determine the average all-to-all broadcast delay. Collected data was subjected to polynomial regression analysis, resulting in a third-degree polynomial approximation of the delay function, which depended on the number of nodes and graph diameter. The Flooding Consensus algorithm was employed to model and calculate the expected value of consensus delay in the presence of node crashes.

CONTRIBUTIONS OF THE DOCTORAL THESIS

1. A 2D, extensible, real-time simulation platform for cooperative UAV missions was developed, supporting multiple control modes and designed for stress testing system resilience under conditions such as network instability, node crashes, and delayed communication. The platform is modular and can be reused as a basis in practical solutions.
2. A method for quantifying the impact of eventual algorithms using Key Performance Indicators (KPIs) was proposed and applied. This approach allows for a direct comparison of algorithm performance based on measurable metrics.
3. The performance of a perfect leader detection algorithm with a conservative timeout was compared to an eventually perfect leader detector in the context of a cooperative UAV perimeter patrol mission, providing data on their reliability and convergence behavior.
4. A Closed Gordon-Newell Queueing Network (QN) model was developed to assess the trade-off between centralized AI service response time and onboard AI accuracy in cooperative UAV patrol missions. The model allows for a direct comparison of system performance under different configurations.
5. A custom probabilistic UDP and HWMP-based all-to-all broadcast protocol was designed and simulated in the NS-3 environment. Simulation results were used to model the behavior of a Flooding Consensus algorithm under conditions of missing communication links.

MAIN CONCLUSIONS OF THE DOCTORAL THESIS

1. When measuring the impact of fault-tolerant algorithms, it is advisable to first develop optimal decision logic, as this significantly influences mission KPI, even in scenarios with extensive faults.
2. Real-time simulation is invaluable for identifying bugs that would otherwise appear during real-world deployment, where fixing them can be significantly more costly. The primary drawback of real-time simulation is its execution time. A hybrid approach—using discrete simulation for overall performance testing and real-time simulation for advanced corner cases in later project phases—can help achieve maximum fault tolerance and resilience.
3. Simulation data indicates that the cumulative number of false crash detections caused by the eventual leader detector algorithm during simulation follows a logarithmic pattern. As the number of deployed UAVs increases, the time until the last false crash detection also increases significantly. With more than five UAVs and a larger detection timeout (DT), false crash detections can continue for up to 10 minutes. Nevertheless, due to the logarithmic nature of this behavior, most false crashes occur quickly, and only a small portion of total false negatives affect KPIs.
4. In cooperative UAV perimeter patrol mission simulations, quantitative KPI metrics are valuable for comparing leader election algorithm performance and for determining whether eventually consistent algorithms are suitable for cooperative UAV scenarios.
5. In the "Perfect Leader Detector" algorithm, extending the crash detection timeout from 15 seconds (ideal) to 60 seconds results in a 12.1% KPI reduction during a five-minute simulation with a 30% UAV crash rate. The "Eventually Perfect Leader Detector," configured with a 5-second base crash detection timeout and a 3-second delta, shows a 4.1% KPI loss compared to the "Perfect" algorithm with the ideal timeout. Notably, the "Eventually Perfect" algorithm outperforms the "Perfect" algorithm under realistic conditions (60s timeout), achieving a 9.7% KPI improvement at a 30% crash rate.
6. A UAV surveillance network employing AI-based threat detection can be modeled using a Closed Gordon-Newell Queueing Network. While improvements in onboard AI accuracy reduce time lost to false positive monitoring linearly, reducing the central AI's response time has a greater impact on minimizing lost monitoring time under increasing system load.
7. The Byzantine error model is widely considered in avionics and space engineering. However, this term is never mentioned in ETSI ITS technical reports. A literature review of ITS reveals that Byzantine fault-tolerant algorithms proposed for V2X systems primarily focus on privacy, blockchain-based architecture, and reputation mechanisms.
8. A custom, randomized, UDP-based, reliable, all-to-all broadcast algorithm using 802.11s mesh, tested with NS3 simulation, shows an average delay of 280ms for 33

nodes with a connectivity graph diameter of 10. For smaller diameters and node counts, average delays are below 100ms.

9. Mathematical modeling shows that the fail-stop flooding consensus algorithm, based on the aforementioned broadcast algorithm, can theoretically achieve consensus in under 500 ms for D=8, N=11, with up to 30% node crashes, and under 200ms for smaller diameters. The shape of the reliable broadcast delay curve depends on the communication graph's diameter: for D=2, delay increases with node count, while for $D \geq 7$, the growth rate decreases. A more linear trend is observed around D=4.

PRACTICAL VALUE OF THE DOCTORAL THESIS

- Developed UDP NAT traversal software to establish autonomous UAV networks within cellular networks. Conducted field experiments to test NAT traversal across various cellular network providers.
- Designed and implemented a UDP-based fault-tolerant communication middleware for cooperative UAV missions. Built a 2D, real-time, extensible simulation framework for cooperative UAV missions, focusing on fault tolerance testing.
- Discovered through simulations that the convergence time of the eventual leader detector algorithm exhibits a logarithmic relationship with the delta parameter and the number of UAVs. Identified the relationship between mission KPIs and the number of UAVs through simulation analysis.
- Analyzed how different configurations of eventual and perfect leader detector algorithms impact mission KPIs, offering guidance for selecting suitable solutions for cooperative UAV missions.
- Developed a mathematical model to inform the design of software and hardware setups for cooperative UAV networks.

The results of the scientific research of the doctoral thesis can be used for the development of cooperative UAV and other mobile node mission and service solutions.

THESES TO BE DEFENDED IN THE DOCTORAL DISSERTATION

1. Using a novel real-time 2D simulation of a cooperative UAV-based perimeter threat search system, with a UDP-based communication stack and an unstable network modeled using log-normal delays ($\mu = 6$, $\sigma = 1.5$), it is possible to evaluate the efficiency of leader election strategies. Experiments show that eventual election with an adaptive crash detection timeout improves mission KPI by 9.7% compared to static conservative timeouts.
2. Using a Closed Gordon–Newell Queueing Network model of cooperative UAV threat search missions, it is possible to show that reducing the response time of the remote AI service minimizes wasted UAV time from misclassifications more effectively than improving onboard AI accuracy.

3. Using a novel NS-3 simulation and a UDP-based probabilistic reliable broadcast, experiments show that the fail-stop flooding consensus algorithm can achieve consensus in 99% of cases in under 500 ms on an 802.11s mesh with 8 hops and 11 nodes, with up to 30% node crashes, and under 200 ms for smaller diameters.

The volume of the doctoral thesis is 100 pages. The thesis consists of five chapters, a conclusion, a list of references, and five appendices.

The first section begins with a brief introduction to distributed systems theory, covering its history, fault tolerance, fundamental building blocks, and complexity. It then explores the phenomenon of how applications and theories have evolved over recent decades, naturally leading to the emergence of mobile network-based distributed systems. The section emphasizes the critical importance of these systems in the modern world, highlights the significantly higher complexity of mobile networks compared to conventional IT systems, and discusses the paradox of low-quality fault tolerance implementation in standard engineering practices. Finally, the section provides all the necessary foundations for the following sections, including a review of existing literature and the problem formulation.

The second section is entirely devoted to the simulation developed for the main experiments of this thesis. It begins by discussing the requirements defined specifically for this research, covering both functional and non-functional aspects, as well as exploring options for reusing existing simulation platforms. Next, it addresses communication-related topics, including NAT traversal, communication protocols, and network delay simulation. The section then delves into the system architecture, focusing on layering and modularity aspects. Finally, it presents the testing strategy, statistical analysis, and conclusions.

The third section focuses on the problem of leader election algorithms in the context of cooperative UAV perimeter patrol. Specifically, two algorithms are analyzed and compared: perfect leader detection and eventual leader detection. The primary objective is to quantify the impact of using the eventual algorithm on mission performance. Various experiments are conducted using the simulation platform described in the previous section. The results are presented with an emphasis on the effects of the eventual algorithm's convergence time and identifying which algorithm performs better in specific scenarios.

The fourth section focuses on a specific aspect of cooperative UAV patrol — the use of a hybrid AI threat detection approach and how improvements to each component impact mission KPIs. It presents the proposed Closed Gordon-Newell QN model, detailing the system components, parameters, and the approach used to calculate the response time probability distribution.

The fifth section addresses the smart vehicle emergency consensus problem and examines whether the Byzantine error model is applicable in V2X scenarios. It begins with a survey of existing literature on the application of the Byzantine model in both V2X and other domains. The second part focuses on calculating the fail-stop flooding consensus execution time in emergency crash avoidance V2V scenarios. It provides a detailed description of the NS-3 simulation setup, which includes an all-to-all 802.11s mesh network and UDP-based

probabilistic reliable broadcast. The broadcast execution time results are approximated and used to model the consensus execution time.

The conclusion of the dissertation summarizes the main findings of the work and defines future research directions. The appendices include lists of conferences, publications, and projects, present the results obtained in experimental measurements and provide the simulation codes.

2. COOPERATIVE UAV PATROL SIMULATION

The goal of this chapter is to comprehensively present the design and development of the mission functionality and its corresponding real-time simulation framework. Moreover, the functionality was developed with reusability in mind, allowing it to be integrated into actual UAV software. As such, it extends beyond the core mission logic and includes components such as a rendezvous server to reduce manual network configuration and enable peer-to-peer communication between UAVs over cellular networks. This chapter serves as a prerequisite for the subsequent chapters, which will detail the experiments conducted using the simulation environment introduced here. This chapter serves as a prerequisite for the subsequent chapters, which will detail the experiments conducted using the simulation environment introduced here.

2.1. Introduction and goals

The primary goal is to develop an extensible and efficient platform for experimenting with various cooperative UAV scenarios. As previously noted, software systems that focus solely on implementing core functionality often neglect negative scenarios and edge cases, leading to reduced resilience and reliability. In distributed systems—particularly those involving mobile cooperative agents—the likelihood of faults increases exponentially across all levels. Therefore, the main focus of the simulation is to provide a framework for testing the system's fault tolerance, with the aim of enabling deployment to real-world infrastructure with minimal friction.

COOPERATIVE UAV PATROL

The following section describes the use case and core functionality of the system (fig. 2.1). The scenario assumes the presence of large facilities, typically located in suburban or remote areas. These may include factories, power stations, airports, ports, strategic infrastructure, military sites, or other high-value assets. A typical perimeter length ranges from 1 to 20 kilometers. The primary objective is to detect potential threats appearing along the perimeter in real time. A secondary objective is to collect as much information as possible about each identified threat—using video feeds and other onboard sensors—to support rapid assessment and response.

Conventional security methods for protecting such facilities are well established and typically include static video surveillance systems and periodic human patrols, often supported by ground vehicles. An emerging and increasingly popular alternative is autonomous video surveillance using cooperative UAVs. This approach offers several advantages, including reduced operational costs, increased effectiveness, elimination of human risk, and adaptability to various types of terrain.

The central idea is that a fleet of UAVs—potentially up to 50 units—can be deployed to autonomously partition the surveillance area and continuously monitor for potential threats. While the specific hardware and infrastructure required to support such a mission are beyond

the scope of this research, several general assumptions are made. Each UAV is equipped with onboard AI capable of real-time object classification. Additionally, a centralized processing unit is assumed to be available for more advanced AI-based analysis, allowing for higher-confidence threat detection and classification.

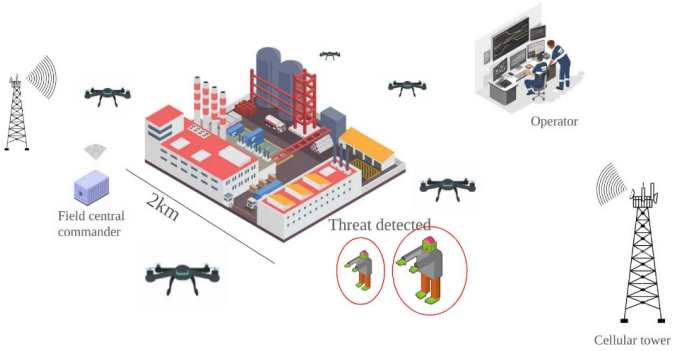

fig. 2.1 Cooperative UAV patrol solution concept.

Also beyond the scope of this research is the infrastructure required to support continuous UAV patrol operations. It is anticipated that autonomous charging and docking stations would be used to enable long-term deployment. Cellular communication is assumed as the default method for UAVs to communicate with each other and with ground systems. The rendezvous process and NAT traversal mechanisms will be discussed in a later chapter.

Additional infrastructure requirements may vary depending on the chosen system architecture. For example, in a centralized architecture, a central server is required to coordinate UAV actions and maintain global state. In contrast, a decentralized architecture may distribute decision-making across the UAVs themselves, requiring different communication and synchronization mechanisms.

The core principle guiding this work is that reliability and security must be built into the system architecture from the outset through fault-tolerant design. While different architectural approaches will be discussed later in this section, the focus of this research is on a *leader-based* architecture. In this model, one UAV is elected as the leader among all participating units.

The leader is responsible for: collecting location data from follower UAVs; processing event reports (such as threat detection or disappearance); calculating the optimal mission strategy; decomposing it into specific tasks; and multicasting those tasks to the appropriate UAVs for execution. For example, the leader must divide the perimeter into segments and assign each segment to the most suitable UAV for patrol.

The leader-based architecture strikes a balance between fault tolerance and mission efficiency. However, as with any UAV, the leader itself is susceptible to failure. In such cases, the remaining UAVs must elect a new leader. The challenge lies in ensuring that all UAVs agree on the same leader—otherwise, the mission may become fragmented or inefficient. This constitutes a classic *consensus* problem, previously introduced, and like all distributed algorithms used in this context, the leader election mechanism must be fault tolerant.

When a threat is identified by a UAV, an event is sent to the leader for evaluation. The leader then determines whether sufficient resources are available to assign two UAVs for tracking the threat. If resources are limited, a single UAV will be assigned to monitor the threat until it disappears for any reason. Threats are assumed to appear randomly along the perimeter. Therefore, the overall mission objective is to maximize threat detection, monitoring, and tracking time, while maintaining fault tolerance in the presence of potential failures—such as UAV crashes, software bugs, or communication delays.

SIMULATION TYPES

One type of simulation implemented in this research is fully deterministic (chapter 5.3). It operates by using a predefined seed, which allows the simulation to unfold all subsequent actions in a fixed, predictable order. For example, with a given seed X, the simulation may generate the following sequence of events: (1) UAV1 broadcasts event A; (2) UAV4 receives the event and generates event Y; (3) UAV7 receives Y and produces event Z; (4) UAV4 broadcasts event Y again; and so on.

This deterministic approach offers three key advantages:

- It guarantees reproducibility — running the simulation with the same seed will always result in the same sequence of actions and outcomes.
- The simulation can be executed significantly faster than real time, limited only by available hardware resources. For instance, it is capable of running up to 100 simulations per minute.
- Debugging is greatly simplified. If unexpected behavior occurs, the exact sequence can be replayed, enabling precise state inspection without the complexity of concurrency mechanisms.

However, this type of simulation also has several limitations:

- The granularity of steps must be defined manually. For instance, messages broadcasted by a node will always execute in the same fixed order. In contrast, real-world systems exhibit significantly greater variability in the timing and interleaving of parallel events, which may expose additional edge cases and potential bugs that the deterministic model cannot capture.
- Integration with real network components is not feasible due to the deterministic nature of the simulation. For example, if the simulation seed generates a successful packet transmission, but the actual network stack experiences packet loss, the simulation would fail to reflect that discrepancy—thus reducing its realism and applicability in testing against real-world conditions.

The second type of simulation is a real-time system. In this mode, the program operates as if actual UAVs are deployed in the field, capturing realistic parallelism and network behavior. This approach enables the simulation of true concurrency, allowing issues that would arise in a real-world deployment to be detected during testing — thereby significantly reducing development and verification costs.

However, real-time simulation is only partially deterministic. While certain elements, such as object generation, can still be controlled through a predefined seed, the behavior of runtime events is influenced by non-deterministic factors such as network latency, scheduling variability, and concurrent interactions. As a result, repeated runs may exhibit slight variations, more closely reflecting real-world system behavior.

For this research, the second type of simulation—real-time simulation—was selected, as it aligns more closely with the primary objective of evaluating system behavior under a wide range of fault-tolerance scenarios. However, this approach has several drawbacks:

- It is not feasible to execute a large number of simulations in parallel, limiting scalability during testing.
- Troubleshooting and debugging are more time-consuming compared to deterministic simulations, due to the presence of real-time concurrency and nondeterminism.
- Writing integration tests is more complex, as the use of real network components introduces variability that is difficult to control and reproduce.

TECHNOLOGICAL STACK

To simulate a UAV network with realistic behavior, the simulation platform must satisfy the following requirements:

1. **Real Network Protocol Support (TCP/IP):** The simulation must utilize real-world networking protocols, such as UDP, to accurately represent inter-UAV communication, including effects such as packet loss, transmission delays, and jitter.
2. **Non-Deterministic Concurrency:** The platform must support parallel processes in a manner that reflects real-world concurrency. Event execution should be governed by real-time scheduling, influenced by system load, network variability, and other dynamic conditions.
3. **Scalability:** The platform should be capable of simulating multiple UAVs and communication nodes while maintaining acceptable performance and responsiveness.
4. **Customization:** It must provide the flexibility to implement specific UAV behaviors, fault-tolerance mechanisms, and environmental models to support a wide range of experimental scenarios.
5. **Repeatability for Debugging:** While the platform must support non-deterministic behavior, it should allow for partial repeatability—such as using fixed seeds for initial conditions (e.g., threat generation)—to aid in debugging and test verification.
6. **Ease of Integration with Hardware (Optional):** If hardware-in-the-loop (HIL) testing is anticipated in future stages, the platform should support integration with real UAV components to enable seamless transition from simulation to deployment.

DETERMINISTIC SIMULATION PLATFORMS

- **OMNeT++**: A modular, C++-based discrete-event simulator primarily used for network simulations. It supports deterministic behavior using fixed seeds and is excellent for modeling and debugging communication protocols. However, it does not support real

non-deterministic concurrency or real-world networking protocols without significant modification.

- **NS-3**: Another discrete-event network simulator with strong support for deterministic modeling of network behavior. It offers UDP and other protocol support but lacks the ability to model non-deterministic concurrency realistically.
- **SimPy**: A Python-based discrete-event simulation library. While easy to use and highly customizable, it does not provide native support for real-time execution or integration with real networking protocols.

PHYSICS-BASED SIMULATION PLATFORMS

- **Gazebo**: A popular robotics simulator that integrates with ROS for high-fidelity physics simulations. While it supports real-time execution and concurrency, its networking capabilities are limited and would require external libraries for UDP support.
- **AirSim**: A Microsoft platform for simulating drones and autonomous vehicles in realistic environments. It focuses more on sensor simulations and does not natively support real networking protocols like UDP.

REAL-TIME AND NETWORKING-CAPABLE PLATFORMS

- **ROS (Robot Operating System)**: A middleware framework designed for robotics. ROS provides tools for real-time execution and concurrency, as well as plugins for UDP communication. However, it requires additional effort to simulate environmental factors and is primarily focused on hardware integration.
- **Custom Docker-Based Simulation**: Using containers, you can simulate multiple UAV instances, each running its own software stack. Real UDP networking and non-deterministic concurrency can be modeled using container networking. This approach provides maximum flexibility but requires significant development effort to build and maintain.

Docker was selected as the base platform for executing individual UAV nodes. It provides maximum flexibility while supporting real networking and protocol implementations, allowing the developed software to be deployed directly on any system running a Linux-based operating system (Table 2.1).

PROGRAMMING LANGUAGE

The choice of Docker as the foundation for networking offers significant flexibility and enables the use of real network protocols. However, it provides limited options for simulating mission physics. While physics libraries could potentially be integrated using specific language runtimes, the decision was made to keep physics modeling minimal and focus primarily on mission logic. All other aspects of the simulation are implemented in C# using the .NET 8 framework.

The selection of .NET is justified by its cross-platform support and compatibility with various operating systems, including embedded platforms such as Raspberry Pi OS. Modern

versions of .NET (e.g., .NET 8) offer support for ARM-based processors, making it suitable for resource-constrained environments typically found in UAV systems. Despite traditional concerns regarding the use of virtual machine-based languages for embedded development, .NET provides several advantages that align with the needs of UAV mission logic.

Specifically, .NET supports multithreading, asynchronous programming, and high-level abstractions for networking (e.g., UDP and TCP), which are essential for implementing distributed UAV coordination protocols. Its modular architecture also facilitates the creation of reusable components for navigation, communication, and fault tolerance—an asset for iterative development and research scalability.

While real-time constraints may present challenges, these can be addressed through hybrid system architectures, where time-critical tasks are delegated to dedicated microcontrollers. Additionally, .NET offers robust tooling for debugging, testing, and integration with machine learning frameworks, further enhancing its suitability for research-oriented UAV applications.

Nonetheless, its applicability on resource-constrained UAV platforms must be carefully evaluated in comparison to lightweight alternatives, particularly in latency-sensitive or high-reliability scenarios.

Table 2.1

Comparison of the simulation frameworks

Feature/Requirement	*OMNeT++*	*NS-3*	*SimPy*	*Gazebo*	*AirSim*	*Custom C# & Docker*
Real UDP Protocol Support	Using plugin	No	No	No	No	Yes
Non-Deterministic Concurrency	No	No	No	No	No	Yes
Scalability	High	High	Medium	Medium	Low	High
Customizability	High	High	High	Medium	Medium	Very High
Repeatability	Yes	Yes	Yes	Yes	Yes	Partial
Ease of Integration with Hardware	No	No	No	Yes	No	Yes

2.2. UAV communication

There are several communication protocol options available for coordinating cooperative UAVs. In the envisioned real-world deployment, cellular communication is selected as the primary medium, for several practical reasons. First, cellular networks provide extensive coverage, including in many rural and remote areas. Second, they offer a ready-to-use infrastructure with high throughput, eliminating the need for dedicated communication systems. To initiate a mission, the operator must configure each UAV with a unique identifier and an associated SIM card. However, a key limitation of cellular networks is that they do not support

direct device-to-device communication by default, due to the absence of public IP addresses for client devices. While many studies treat the cellular network as a primary or supplementary communication channel, few delve into the challenge of establishing P2P communication in this context [36] [37] [38]. Cellular infrastructure is not designed for each device to act as a server. Despite this limitation, many smartphone applications—including messaging platforms—successfully enable peer-to-peer communication. This is achieved through the use of NAT traversal techniques, which allow devices behind private IPs to establish direct communication channels (fig. 2.2). These same principles can be applied to enable UAV-to-UAV communication over cellular networks.

NAT TYPES

There are several classes of NAT devices, ranging from less restrictive to highly restrictive, each affecting the feasibility of P2P communication in different ways.

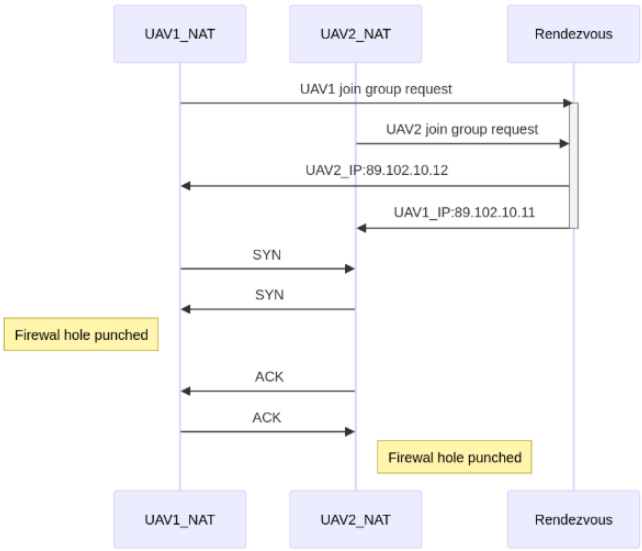

fig. 2.2. UDP NAT traversal algorithm sequence diagram.

Full Cone NAT is the least restrictive type. All incoming packets from a specific external IP address and port are forwarded to the same internal IP and port, as long as the NAT mapping remains active. This behavior enables straightforward inbound communication, making Full Cone NAT ideal for P2P applications.

Restricted Cone NAT imposes more constraints. It allows incoming packets from any external host, but only if that host has previously received a packet from the internal device. While more restrictive than Full Cone NAT, it still permits hole punching and P2P communication, provided the internal device initiates contact.

Port-Restricted Cone NAT adds another layer of restriction. It behaves similarly to Restricted Cone NAT but enforces that the external host must also match the specific port to

which the internal device had previously communicated. This makes P2P connection setup more challenging, although NAT traversal techniques like UDP hole punching can still be effective under certain conditions.

Symmetric NAT is the most restrictive and complex. It generates a unique mapping for each combination of internal source and external destination address and port. As a result, the internal device appears to use a different public port for each external peer. This behavior complicates NAT traversal significantly, making inbound communication from peers nearly impossible unless explicitly initiated and maintained by the internal device.

Understanding the behavior of these NAT types is crucial when designing P2P communication mechanisms for cooperative UAV systems operating over cellular networks.

NAT TRAVERSAL

STUN (Session Traversal Utilities for NAT) is a widely adopted technique that enables devices behind a NAT to determine their public-facing IP address and port, thereby facilitating peer discovery and direct communication. However, STUN is not always effective, particularly when dealing with restrictive NAT types such as symmetric NAT, which often block incoming traffic unless it is part of an explicitly established connection.

To address these limitations, hole punching is commonly used in conjunction with STUN. Hole punching allows peers to create temporary NAT mappings by exchanging messages, enabling direct communication paths. While this method is effective in many scenarios, it may still fail under certain network conditions. In such cases, TURN (Traversal Using Relays around NAT) is used as a fallback mechanism. TURN relays traffic through a third-party server, ensuring connectivity when direct P2P communication is not feasible.

Additional NAT traversal techniques include UPnP (Universal Plug and Play), which can automatically configure NAT devices to open specific ports, and ICE (Interactive Connectivity Establishment), which combines STUN and TURN to test and select the most effective communication path. It is important to note that some NAT types, especially symmetric NAT, are incompatible with STUN alone, requiring a combination of the above methods for successful traversal.

In the context of this research, the only viable approach is to use UDP hole punching, with a relay server (TURN) as a fallback [39]. Although hole punching is theoretically possible for UDP, TCP, and ICMP protocols, ICMP is not suitable for inter-UAV communication, and TCP hole punching is significantly more complex due to the stateful nature of the TCP protocol. Consequently, UDP remains the most practical and reliable option.

UDP HOLE PUNCHING EXPERIMENTS IN THE FIELD

Several experiments were conducted to evaluate the feasibility of UDP hole punching under real-world conditions using different cellular network operators. The objective was to determine whether direct peer-to-peer UDP communication could be established between clients operating behind NATs in mobile networks.

Experiment 1 involved a configuration with one LMT (Latvian Mobile Telephone) cellular client and one Azure cloud server with a public IP address. This setup was successful, confirming that outbound UDP traffic from the LMT client could reach the public server and receive responses, with NAT mappings persisting long enough to enable stable communication.

Experiment 2 tested direct communication between two LMT cellular clients located within close physical proximity. Despite both devices receiving distinct public IP addresses from the operator, the attempt failed. This result suggests that NAT or firewall policies applied by LMT restrict peer-to-peer UDP connectivity—possibly due to symmetric NAT behavior or additional security mechanisms implemented by the operator.

Experiment 3 paired one LMT client with one BITE cellular client. Under similar conditions to the previous experiment, UDP hole punching was successful. This indicates that at least some combinations of mobile network operators support direct UDP communication, likely due to differences in NAT configuration or firewall policies between providers.

As an alternative to NAT traversal, some cellular network operators offer SIM cards with publicly routable IP addresses for an additional fee. This approach simplifies p2p communication and also enables the use of TCP, which may be preferable in scenarios requiring reliable delivery and connection state management.

NETWORK STATE SIMULATION

Simulation of network delays, packet drops, and network partitioning is a core component of the system (fig. 2.3). The following functional and non-functional requirements are defined for this module:

- **Reproducibility:** The module must support reproducible delay, drop, and partitioning sequences based on a seed parameter. Since delays and packet loss are primary sources of concurrency-related issues in real-world systems, it is essential to simulate the same sequence consistently for testing and debugging purposes.
- **Configurable Models:** The module must allow for customizable probabilistic models and packet drop scenarios to represent various real-world network conditions.
- **High Performance:** The simulation should not introduce any additional delay beyond the configured values and must be capable of handling hundreds of requests per second without degradation in performance.
- **Correct Routing:** The proxy must be able to correctly route response packets back to the original sender, maintaining end-to-end communication integrity.

Theoretically, there are two approaches to simulating network delays in a distributed network of Docker containers. The first is a **decentralized approach**, in which each sender is responsible for calculating and applying delays locally. This method offers several advantages:

- No need to manage a separate proxy server.
- No requirement to design a custom API for sending packets through a proxy.
- Slightly more deterministic behavior, as delay sequences are distributed across UAVs. For example, if UAV X experiences delay pattern Y in simulation Z_1, the same

combination will be present in simulation Z_2. In contrast, a centralized approach may introduce variability due to concurrency.

However, the decentralized approach has notable drawbacks:

- If a UAV container becomes overloaded, it may negatively impact the performance and timing accuracy of the network simulation.
- Implementing complex scenarios such as network partitioning is difficult, if not infeasible, in a decentralized setup.

For this project, the centralized approach was selected, as it provides better performance and enables full control over simulation scenarios—particularly in modeling advanced conditions like network partitioning and synchronized delay patterns.

To facilitate the transmission of UDP packets via a proxy, a custom minimal protocol was developed. The UAV is configured with the proxy's address. Each UDP message is divided into two sections: a payload and a header. The header contains the original destination address of the packet.

Upon receiving a packet, the proxy extracts the original destination address, applies any configured network modeling pipelines, and then forwards the packet to the intended destination. Additionally, the proxy appends a header containing the original packet source information. This allows the recipient to extract the source address and send a response using the same proxy-based approach.

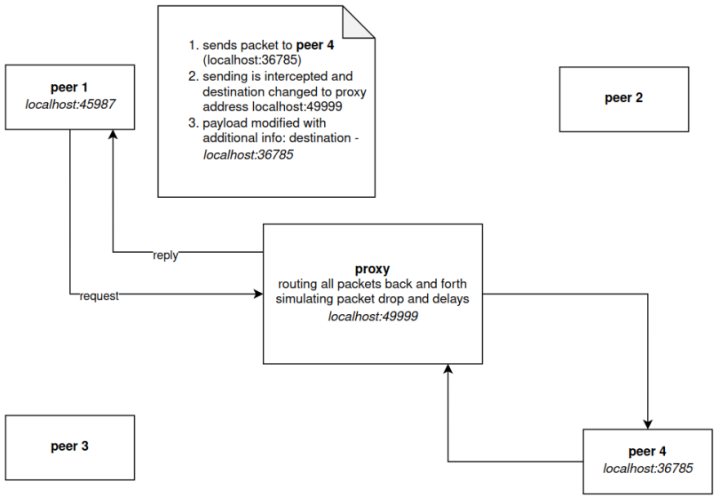

fig. 2.3. Network proxy architecture.

THROUGHPUT

To evaluate the performance limits of the UDP proxy server, its throughput was measured in terms of both packets per second and bytes per second. This dual-metric approach was essential due to the system's varying operational modes—particularly under leaderless consensus scenarios, which generate a high volume of small messages. In such cases, measuring only bandwidth (bytes per second) would provide an incomplete assessment. Therefore, special

emphasis was placed on accurately capturing the maximum number of packets the proxy could process per second.

The experiments were conducted locally on a standard laptop, providing a controlled environment free from external network variability. The proxy was subjected to a steady stream of UDP packets generated by a dedicated client script transmitting at a fixed rate. A corresponding receiver was configured to count and validate incoming packets routed through the proxy. Throughput was determined by running multiple trials with incrementally increasing message rates until packet loss occurred. The highest stable rate achieved without drops was recorded as the proxy's throughput ceiling.

The primary metric obtained was packet throughput, which reached approximately 350 UDP packets per second. This figure represents the practical upper limit of the proxy's ability to process and forward messages under load, accounting for internal processing delays.

While the focus was on packet rate, data throughput in bytes per second was also estimated. Packet sizes in typical simulation scenarios ranged from 50 to 250 bytes. Based on this, the estimated bandwidth utilization lies between 17,500 bytes per second (350 packets/sec × 50 bytes) and 87,500 bytes per second (350 packets/sec × 250 bytes), providing a realistic view of the proxy's performance across various message sizes.

2.3. Simulation architecture

The simulation platform is developed in C# using the .NET 8 framework and is structured as a modular system designed to evaluate UAV collaboration and communication in dynamic threat environments. The architecture prioritizes minimalism and efficiency, supporting both single-process execution—for debugging and analysis—and distributed deployment using Docker, which facilitates scalability and memory isolation.

At the core of the platform is the **World** module (fig. 2.4), which simulates a two-dimensional virtual environment. This module is responsible for updating UAV positions, injecting threats, and recording the simulation state on a frame-by-frame basis. It also produces a simplified video stream, where each frame is rendered using a UTF-16 character-based grid, enabling lightweight visualization directly in the console. This same 2D symbolic representation is used to emulate UAV vision, wherein each UAV perceives its environment through a localized window rendered as a matrix of UTF-16 symbols.

UAVs are modeled as autonomous agents. In single-process mode, all UAVs execute as threads within a single executable, whereas in Docker mode, each UAV runs in an isolated container. The latter configuration supports memory isolation and better simulates distributed processing environments. Each UAV interacts with the simulation by issuing actuator commands and receiving simulated sensor data—such as GPS coordinates and vision frames—from the **World** module.

Inter-UAV communication is a critical component of the simulation and is implemented using UDP sockets. A **Network Simulator** module introduces realistic communication constraints by injecting packet loss and latency according to configurable probability

distribution functions (PDFs). This enables the evaluation of control and coordination algorithms under non-ideal network conditions.

For simulations involving NAT or real system testing in the field, a **Rendezvous/NAT Traversal Server** is included. This server broadcasts the active IP addresses of UAVs and coordinates UDP hole punching, thereby simulating environments in which UAVs are connected through NATs or firewalled networks.

Each UAV is connected to a **Stats/KPI Calculation** library, which computes key performance indicators such as packet throughput and threat registration efficiency. Throughput is measured in both messages per second and bytes per second, while threat-related metrics are derived based on how effectively UAVs detect and sustain tracking of threats. All statistics are recorded as structured output files to support further analysis.

Simulation execution is initiated through a **Console Launcher**, which parses input parameters and launches the necessary modules and services. Depending on the selected mode, this may either start a monolithic simulation instance or deploy a **Docker Compose** configuration in which each UAV operates in a separate container with an isolated IP address (e.g., 172.18.0.X). This containerized deployment ensures a deterministic and reproducible environment, facilitating scalability and controlled experimentation.

All simulation output is written to a **Simulation Output File**, which can later be visualized using a **Simulation Recording Visualizer**. This tool replays the simulation using the same UTF-16 console-based rendering engine, allowing for efficient post-experiment inspection without the need to re-run the full simulation.

This architecture strikes a balance between performance and simplicity, enabling repeatable experiments across a wide range of network, sensor, and algorithm configurations. The use of a 2D environment with character-based rendering supports efficient prototyping, particularly in resource-constrained settings. At the same time, Docker integration facilitates scalability, making it suitable for larger, distributed experimental deployments.

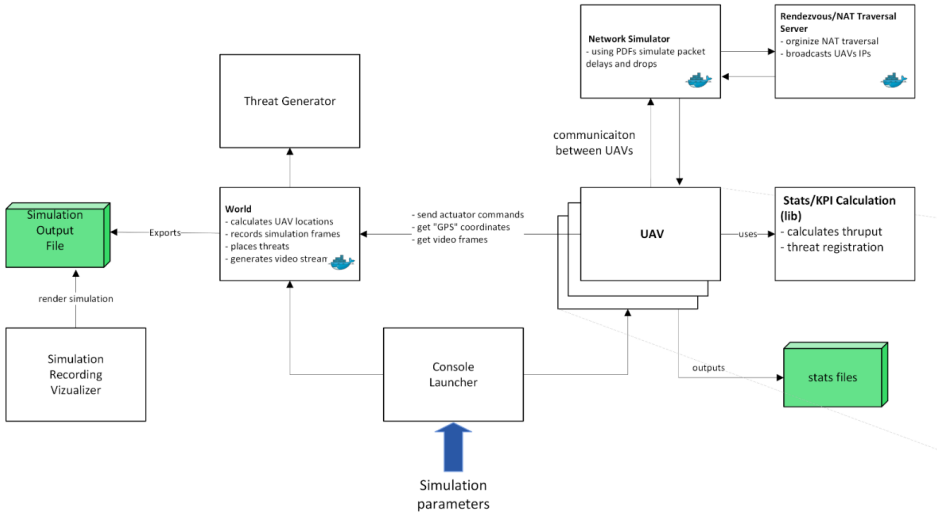

fig. 2.4. Architecture of the simulation platform.

UAV ARCHITECTURE

Each UAV is implemented as a modular .NET 8 component, capable of running either as a thread or in its own Docker container. The architecture is divided into the following main components: control logic, task scheduler, physical task executors, in-memory database, actuators and sensors, communication middleware, and event routing.

The first component is the UAV's brain, called **DroneCore**—a module responsible for responding to all types of inputs received by the UAV. These inputs may include location updates from slave UAVs, notifications of detected or cleared threats, heartbeat requests, or task assignments from the leader. Responses may include local database updates, execution of physical tasks, outgoing messages, or any combination of these. The **DroneCore** module is composed of pure functions (in the functional programming sense), which do not cause side effects. This design separates UAV logic from infrastructure and "glue" code, improving code quality and maintainability.

The next component is the **scheduler**, which handles periodic tasks such as sending heartbeat messages, broadcasting location, retrieving sensor data, retrying message transmissions, and polling for physical task updates.

UAVs must also perform real-world tasks—such as patrolling designated areas and tracking threats. While **DroneCore** calculates mission plans, these plans must be translated into actual actuator commands. This is handled by two components: **PatrolTaskExecutor** and **ThreatTracker**. Commands to these modules usually originate from the leader UAV. It is important to process these messages before forwarding them to the executors, as they may arrive out of order.

An in-memory database is used to store the UAV's operational state. For example, a leader UAV keeps track of other UAVs' coordinates and their most recent mission states—such as which UAV is performing which task. If leadership changes, the new leader will take over responsibility for persisting this data.

Finally, the system supports several mission control modes: **leader-based**, **centralized**, **decentralized with consensus**, and **decentralized without consensus**. These modes are explained in more detail in Section 3. From an implementation perspective, the leader-based mode is a superset of the centralized mode—it not only manages mission state and aggregates events from subordinates but can also dynamically elect a new leader. In contrast, the decentralized modes do not rely on a single decision-maker. Each UAV computes mission state independently. The two decentralized modes differ in how they coordinate: one uses a total-order broadcast (consensus) algorithm before each event, which is inefficient but ensures consistency. The other operates without consensus; instead, each UAV broadcasts all information it holds.

COMMUNICATION MIDDLEWARE

One of the core layers of the solution is the communication middleware. This layer is designed to abstract the complexity of the fault-tolerant algorithms and provide a convenient API to the higher application layer of the UAV. The following methods are available for the UAV in different control modes:

- **SendToGroup:** Implements a best-effort broadcast algorithm.
- **SendTo:** Provides a reliable link using retries, unique message IDs, and acknowledgments.
- **JoinChat:** Establishes p2p connections via a Rendezvous server using UDP hole punching.
- **FailureDetectors:** Implements both eventual and perfect failure detector algorithms as described in section 3.1.
- **FailStopFloodingConsensus:** Implements a fail-stop (synchronous) consensus algorithm, used in decentralized control mode without a leader.
- **ReliableBroadcast:** Supports two types—eager and lazy reliable broadcast, as defined in [1].
- **AtomicBroadcast**: Implements total order broadcast using flooding consensus as the underlying mechanism. Primarily used in decentralized control mode without a leader.

DECISION LOGIC (DRONE CORE)

All decision-making in the simulation is handled by a group of simple, focused classes within the DroneCore. PureLogic module. These classes are entirely decoupled from sensors, actuators, or networking. Instead, they operate solely on the current state of the world and produce decisions—such as which UAV should perform which task. This separation of concerns makes the logic highly testable, reusable, and easier to maintain without impacting the rest of the system.

The central class in this module is GlobalMissionCalculator. It evaluates the current positions of all UAVs, the list of active threats, and the mission area boundaries, then assigns UAVs to two primary roles: **threat tracking** and **perimeter patrolling**.

For threat tracking, the ThreatControlTaskCalculator is used. It aims to ensure that each threat is monitored by one or two UAVs. The algorithm first assigns free UAVs to untracked threats, prioritizing assignments based on proximity. Then, it checks whether any threat is currently tracked by only one UAV and, if possible, assigns a second one for redundancy. One UAV—typically the one farthest from all threats—is intentionally left unassigned at this stage and is later designated for patrolling.

The PatrolAreaCalculator manages any UAVs not involved in threat tracking. It divides the perimeter into patrol zones, distributing UAVs evenly across them and introducing slight overlaps between zones to avoid blind spots. Each zone is assigned to the nearest available UAV.

Several smaller utility classes support the main logic. PatrolTaskExecution determines how a UAV should enter and navigate its assigned patrol zone. ThreatProcessingLogic answers questions such as whether a threat is still being tracked. LeaderElection selects a leader UAV by choosing the one with the lowest ID from a sorted list. All spatial calculations, such as distance and heading, are delegated to the TwoDHelpers class.

Importantly, all logic within the PureLogic module is stateless and deterministic. It holds no internal state between calls, ensuring that the same input always produces the same output. This design makes the system predictable, debuggable, and amenable to formal verification.

While the topic of optimal task allocation within a group goes beyond the scope of this paper, relevant theory and practice can be found in sources such as [40].

KEY PERFORMANCE INDICATORS

To evaluate the effectiveness of UAV behavior and coordination strategies, the simulation tracks several key performance indicators (KPIs). These metrics assess how well threats are detected, monitored, and controlled, as well as the efficiency of UAV communication.

- **T.reg. (%) – Threat Registration Rate.** This metric represents the percentage of threats successfully registered by UAVs during the simulation. A threat is considered registered if it is detected by a UAV during patrol. The value is calculated as the number of registered threats divided by the total number of threats generated.
- **T.lifetime mon. (%) – Threat Monitoring Time.** This KPI measures the proportion of threat lifetimes that were actively monitored by UAVs. Each threat has a predefined lifetime, and any time it is monitored by a UAV is counted toward this metric. It reflects the effectiveness of continuous surveillance.
- **T.contr. (%) – Threat Control Rate.** Once a threat is registered, the system may assign two UAVs to monitor it simultaneously—a condition referred to as "control." This metric captures the percentage of threats that, at any point during the simulation, reached this controlled state.
- **T.lifetime contr. (%) – Controlled Lifetime.** This KPI measures the percentage of the total threat lifetimes during which all threats were under control, meaning each had at least two UAVs assigned. It indicates how consistently the system maintained full coverage over active threats.
- **Total Sent (MB).** The total amount of data transmitted by all UAVs throughout the simulation, measured in megabytes. This includes all communication, such as coordination messages and sensor data sharing.
- **Peak Rate (Kb/s).** The maximum data transmission rate observed during the simulation, measured in kilobytes per second. This metric helps identify the peak communication load experienced by the network.

Together, these KPIs provide a comprehensive view of mission performance, balancing threat response effectiveness with communication efficiency. They serve as a basis for comparing different UAV coordination strategies and assessing the impact of network conditions.

INPUT PARAMETERS

On Table 2.2 all possible simulation parameters can be seen.

Table 2.2

Simulation input parameters

Parameter name	*Description*	*Example value*
LOGGING_LEVEL	Level of text-based logs captured from all components (Serilog).	Verbose
WORLD_URL	URL that is used by UAVs to send actuator commands & receive sensor data	http://127.0.0.1:34000
UDP_PROXY_URL	Proxy URL	127.0.0.1:17000
RENDEZVOUS_URL	NAT traversal & UAV initialization server	127.0.0.1:18000
WORLD_FPS	Frequency of world state update	20
SCREEN_UPDATE_RATE	How frequently frames requested by vizualizer	15
DIAMETER	Camera field of view in UTF16 symbols count	20
LENGTH	Perimeter length in meters.	500
WIDTH	Perimeter width in meters	100
DRONE_SPEED_MS	Drone speed in m/s	12
SIMULATION_TIME	Duration of one seed simulation in s	300
DRONE_COUNT	Drone count to be spawned	10
THREAT_PER_MIN	Frequency of threat spawn	5
THREAT_MEAN_LIFETIME	How long in average threat will be displayed (s)	40
DROP_PERCENT	Fraction of UDP packets that will be dropped by proxy (%)	1
WORKER_COUNT	Network proxy thread count	5
MU	Network proxy log normal, MU parameter	6
SIGMA	Network proxy log normal, SIGMA parameter	1.5
RETRY_INTERVAL_MS	Period after which UDP message will be retried if no ACK received (ms)	3500
EVENTUAL_DETECTOR_DT_TOUT	Eventual leader election. Delta amount of timeout that will be increased in cases of false positive detection. (ms)	3000
FAIL_DETECTOR_TIMEOUT_MS	Perfect leader election. Static timeout (ms)	15000
PERCENT_CRASHED	Fraction of UAVs that will crash during simulation (%)	30
SEED	Seed used to generate networks delays, threat locations, etc.	12345
VISUALIZATION	Toggle if visualization enabled (Boolean)	false
FILE_NAME	Folder name where to store simulation files	“Testing eventual convergence time”

TESTING

Significant attention was given to the topic of automated testing for the solution. A well-defined automation testing strategy and implementation enable the incremental delivery of high-quality and complex software over an extended period. Automated tests can be classified into three main categories: (1) unit tests, (2) integration tests, and (3) end-to-end tests. The testing pyramid approach was applied during the current development [testing pyramid].

The primary goal of automated tests is to ensure the stable growth of the software project. An additional benefit of well-designed automated tests is improved software quality. The key attributes of a good, automated test are: (1) protection from regression, (2) refactoring tolerance, and (3) ease of maintenance [41].

Integration and end-to-end tests provide the highest level of bug protection. However, in the current development process, creating such tests proved challenging due to the complexity of test assertion logic — particularly in determining which values to assert, such as UAV locations or received messages. As a result, unit tests were used for modules and functions without out-of-process dependencies, such as schedulers and the network stack.

The project comprises a total of 230 tests, including 30 integration tests, with the remainder being unit tests. The simulation was developed by a single individual over nine months, resulting in the creation of ten C# projects (excluding testing projects), encompassing 381 classes and approximately 32,000 lines of code.

Figure 2.5 below shows a visualization of the simulation along with an example log. Threats are represented using two-symbol UTF-16 labels.

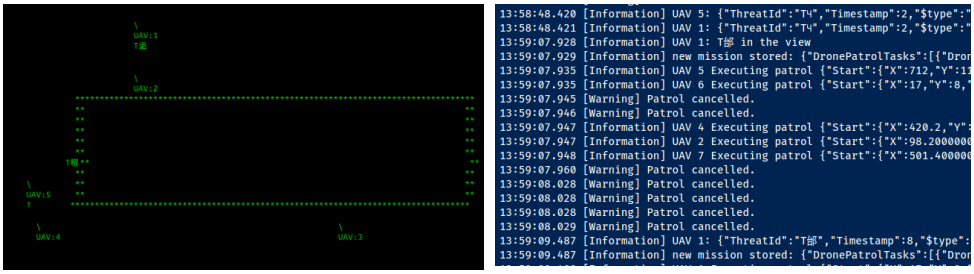

fig. 2.5 Example of the simulation visualizer (left) and real time logs (right).

2.4. Conclusions

The goal of this chapter was to describe the simulation platform developed for testing cooperative UAV missions under realistic fault conditions. The platform supports real-time execution, asynchronous communication, and distributed deployment using Docker containers. It was designed with the specific aim of reusing nearly all of its components—especially the mission logic and coordination algorithms—in real deployments, without the need for rewriting.

Simulation frameworks such as NS-3 are effective for protocol-level experiments but are not well suited for testing system-level behavior, concurrency, or integration with real hardware. In contrast, the approach presented in this work allows the same implementation used in simulation to be transferred almost directly to physical UAV systems, enabling consistent development and testing workflows.

Several key observations were made during the development and use of the simulation platform:

- One of the most critical failures was caused by **internal concurrency issues**, not just network conditions. These were especially visible in the leader UAV when processing bursts of messages. In some cases, tasks were assigned incorrectly or with delays due to race conditions in internal state handling.
- **Real-time execution was essential** for exposing timing-related bugs. These faults would not appear in deterministic simulations, where the order of operations is fixed and timing is controlled.
- The **main drawback of real-time simulation** is the execution time itself. If the simulated mission lasts three minutes, then the simulation takes three real minutes to

complete. This makes it difficult to run large numbers of scenarios quickly and limits statistical analysis.

- Although Docker was originally selected to enforce memory isolation between UAVs, **the actual memory usage per UAV was minimal**. The more important benefit was the ability to simulate real networking behavior and distributed deployment scenarios.
- Writing **automated integration tests** was difficult due to timing variability and asynchronous execution. Fixed seeds allowed some repeatability, but complete determinism could not be achieved in a real-time setting.
- It is not sufficient to model only delay and packet loss. **Network partitions and asymmetric connectivity** also affect system performance. The centralized delay proxy helped to simulate some of these conditions, but more advanced tools would improve accuracy and flexibility.

In summary, the platform described enables realistic testing of UAV coordination strategies and is directly applicable to real-world deployment scenarios. Its architecture supports high code reusability, realistic concurrency, and fault modeling. Future work may focus on extending fault injection capabilities, supporting more complex network scenarios, and integrating physical UAV hardware for hybrid simulations.

3. ANALYSIS OF THE IMPACT OF EVENTUAL LEADER ELECTION ON COOPERATIVE UAV MISSION

This chapter presents the results of research on how eventual leader election influences the performance of cooperative missions. It begins by describing the theoretical foundations and distributed system primitives used in the experiments. Next, the research gap is identified based on a review of existing work in the field. Finally, the problem is formulated, the experimental setup is explained, and the results are discussed along with key conclusions.

3.1. The context

THE GOAL

One key objective is to evaluate whether eventually consistent leader election can be effectively applied in cooperative scenarios such as perimeter patrol. A central challenge is understanding how temporary leadership inconsistencies impact mission performance. To investigate this, mission effectiveness is measured using quantitative metrics (KPIs), enabling an analysis of how inconsistency affects overall mission quality. Another aim is to explore how the duration of the failure detection timeout influences these KPIs, and whether optimizing this timeout—using dynamic or adaptive strategies—can improve performance. Additionally, the study examines the typical duration of temporary inconsistencies and the system parameters that influence them. Finally, the research seeks to clarify how fundamental distributed systems concepts apply to UAV cooperation, including suitable failure models, timing assumptions, and coordination primitives.

MISSION CONTROL MODEL

The authors of the [40] present a comprehensive classification of possible architectures for cooperative agent missions. The figure below (fig. 3.1), inspired by the book, illustrates different types of mission control approaches. The colored cubes represent the control modes supported by the simulation.

Central commander (blue cube) refers to a mode where a single stationary server—typically positioned near the mission area—collects data from all UAVs, computes the optimal task allocation, and multicasts tasks to individual UAVs. If the central commander becomes unavailable (due to network partition, hardware failure, cyberattack, or software issue), the mission is likely to stop. According to the [40], this architecture provides the highest mission quality but suffers from poor scalability.

Consensus on leader (also called *dynamic leader*) is the focus of this research [42]. In this mode, one UAV temporarily takes over the role of the central commander, becoming the leader. The leader is elected dynamically by the UAVs using a leader election algorithm, which will be described in later chapters. From the simulation's perspective, the leader UAV functions identically to a central commander. This approach solves the issue of single point of failure.

Consensus without a leader refers to a mode in which UAVs maintain agreement on the mission state without designating a leader. This requires implementing total order broadcast (explained later), as all events that influence the shared mission state must be broadcasted to all participants. Without a predefined order, events may be processed differently by each UAV. To avoid inconsistencies, this approach requires high message overhead and assumes a synchronous communication model, as leaderless consensus is only possible under synchrony [4].

No consensus, no leader is a mode in which UAVs broadcast events to each other but make decisions independently without coordination. This model is useful for studying the impact of having no agreement in cooperative missions where task sharing is expected. It represents a lower bound on coordination and mission quality.

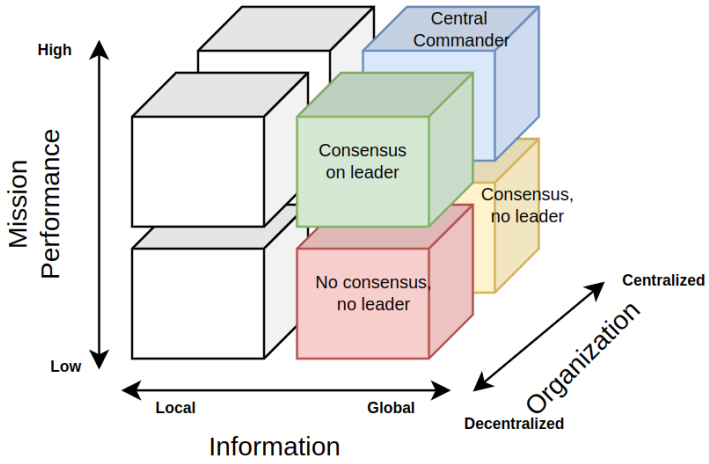

fig. 3.1. Cooperative control mode classification.

COMMUNICATION PRIMITIVES

Each control mode relies on a distinct set of communication primitives. The central commander and leader-based modes utilize what is referred to as a *reliable link* [1], implemented on top of the UDP layer as illustrated in the figure 3.2. This reliable link abstraction is later extended to support reliable multicast communication.

Implementing sequence numbers for messages exchanged between nodes is essential to ensure correct ordering and consistency. In leader-based coordination, tasks may change frequently and could be delivered out of order. For instance, the leader may initially assign patrol task X_1 to a UAV, but shortly afterward revise the decision and issue task X_2 for the same UAV. If task X_1 is delivered after X_2, it must be ignored by the UAV to prevent outdated instructions from being executed.

More complex scenarios may arise where the leader cannot determine, without additional context, which task or event is outdated. For example, UAV 1 might detect a threat X, while UAV 2 later reports that the threat has disappeared. If the leader receives the disappearance

event before the detection event due to message delays, it may erroneously assume the threat still exists and instruct UAVs to track it.

One potential solution is for the leader to maintain the log of commands or events associated with each threat. If the timestamp or sequence number of *threatDetected[X]* is greater than that of *threatGone[X]*, then the disappearance event can be disregarded. This approach ensures that decisions are made based on the most recent and consistent information available.

```
Implements:
    StubbornPointToPointLinks, instance sl.

Uses:
    FairLossPointToPointLinks, instance fll.

upon event ⟨ sl, Init ⟩ do
    sent := ∅;
    starttimer(Δ);

upon event ⟨ Timeout ⟩ do
    forall (q, m) ∈ sent do
        trigger ⟨ fll, Send | q, m ⟩;
    starttimer(Δ);

upon event ⟨ sl, Send | q, m ⟩ do
    trigger ⟨ fll, Send | q, m ⟩;
    sent := sent ∪ {(q, m)};

upon event ⟨ fll, Deliver | p, m ⟩ do
    trigger ⟨ sl, Deliver | p, m ⟩;
```

```
Implements:
    PerfectPointToPointLinks, instance pl.

Uses:
    StubbornPointToPointLinks, instance sl.

upon event ⟨ pl, Init ⟩ do
    delivered := ∅;

upon event ⟨ pl, Send | q, m ⟩ do
    trigger ⟨ sl, Send | q, m ⟩;

upon event ⟨ sl, Deliver | p, m ⟩ do
    if m ∉ delivered then
        delivered := delivered ∪ {m};
        trigger ⟨ pl, Deliver | p, m ⟩;
```

fig. 3.2. “Perfect link” abstraction that is implemented on top of UDP.

The weakest form of broadcast is *best-effort broadcast* (BEB), which provides the following guarantees:

- **BEB1: Validity** — If a correct process broadcasts a message m, then every correct process eventually delivers m.
- **BEB2: No duplication** — No message is delivered more than once.
- **BEB3: No creation** — If a process delivers a message m with sender s, then m was previously broadcast by s.

However, BEB does not prevent disagreement in the case where the sender fails mid-broadcast. In such cases, some processes may receive the message while others do not, leading to inconsistent views across the system. Due to this limitation, best-effort broadcast is not used in our simulation.

One control mode—*no consensus, no leader*—does rely on broadcast, but it requires a stronger abstraction: *regular reliable broadcast* (RB), which provides the following properties:

- **RB1: Validity** — If a correct process p broadcasts a message m, then p eventually delivers m.
- **RB2: No duplication** — No message is delivered more than once.

- **RB3: No creation** — If a process delivers a message m with sender s, then m was previously broadcast by s.
- **RB4: Agreement** — If a message m is delivered by any correct process, then it is eventually delivered by all correct processes.

These additional guarantees ensure consistency even in the presence of partial failures and are critical for modes that operate without centralized coordination or consensus.

A stronger level of broadcast is *uniform reliable broadcast*, which guarantees that the set of messages delivered by faulty processes is always a subset of the messages delivered by correct processes. In other words, no message is delivered by a faulty process unless it is also delivered by all correct ones. This property ensures stronger consistency but is not applied in the UAV simulation, as message delivery by a failing UAV is generally irrelevant to system behavior once it becomes non-operational.

The most complex form of broadcast used in the simulation is *total order broadcast*, also known as *atomic broadcast* [1]. This abstraction is implemented in the "consensus, no leader" mode. It ensures that all correct processes deliver messages in the same total order, requiring synchronization between events and a significant number of messages. Due to its complexity and high communication overhead, total order broadcast is not analyzed in detail within the scope of this research.

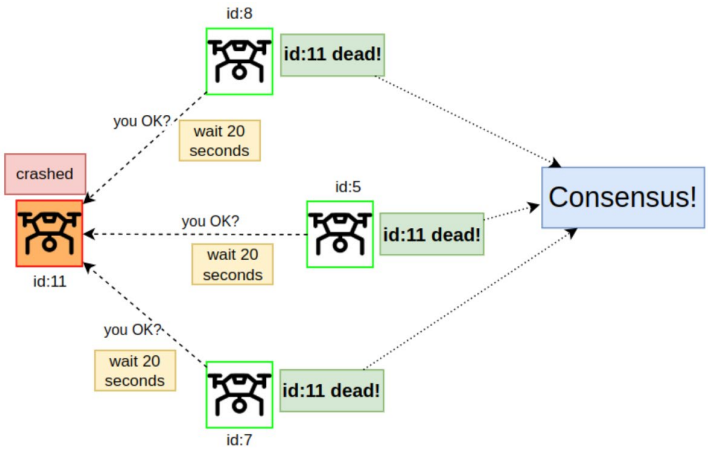

fig. 3.3. Example of the execution of perfect leader election algorithm. All nodes consistently detected crashes, but it took longer 20s delay.

FAILURE DETECTORS

This research employs two types of failure detectors: the *Perfect failure detector* (*P*) and the *Eventual perfect failure detector* (*∗P*), both defined in Section 1. It is important to note that the term "perfect" does not imply that crash detection is synchronized across all nodes. In practice, if the crash detection timeout is *T*, then the time difference between any two nodes detecting the failure of a process *X* will be less than *T*, and the worst-case detection time will be bound by *2T*. This occurs because a node may respond with an acknowledgment at the

beginning of a heartbeat interval, causing nearly $2T$ to pass before its absence is detected. While increasing the frequency of checks can improve detection time, it comes at the cost of higher network traffic, potentially impacting overall system performance.

```
Implements:
    EventuallyPerfectFailureDetector, instance ◇P.

Uses:
    PerfectPointToPointLinks, instance pl.

upon event ⟨ ◇P, Init ⟩ do
    alive := Π;
    suspected := ∅;
    delay := Δ;
    starttimer(delay);

upon event ⟨ Timeout ⟩ do
    if alive ∩ suspected ≠ ∅ then
        delay := delay + Δ;
    forall p ∈ Π do
        if (p ∉ alive) ∧ (p ∉ suspected) then
            suspected := suspected ∪ {p};
            trigger ⟨ ◇P, Suspect | p ⟩;
        else if (p ∈ alive) ∧ (p ∈ suspected) then
            suspected := suspected \ {p};
            trigger ⟨ ◇P, Restore | p ⟩;
        trigger ⟨ pl, Send | p, [HEARTBEATREQUEST] ⟩;
    alive := ∅;
    starttimer(delay);
```

fig. 3.4. Eventually perfect failure detector implementation – Increasing timeout algorithm [1].

LEADER ELECTION

This research considers exactly two leader election algorithms; each derived from one of the previously described failure detectors. For simplicity, we refer to the algorithm based on the perfect failure detector P as the *"perfect"* leader election algorithm, and the one based on the eventual perfect failure detector $*P$ as the *"eventual"* algorithm (fig. 3.7, fig. 3.4).

```
Implements:
    LeaderElection, instance le.

Uses:
    PerfectFailureDetector, instance P.

upon event ⟨ le, Init ⟩ do
    suspected := ∅;
    leader := ⊥;

upon event ⟨ P, Crash | p ⟩ do
    suspected := suspected ∪ {p};

upon leader ≠ maxrank(Π \ suspected) do
    leader := maxrank(Π \ suspected);
    trigger ⟨ le, Leader | leader ⟩;
```

```
Implements:
    EventualLeaderDetector, instance Ω.

Uses:
    EventuallyPerfectFailureDetector, instance ◇P.

upon event ⟨ Ω, Init ⟩ do
    suspected := ∅;
    leader := ⊥;

upon event ⟨ ◇P, Suspect | p ⟩ do
    suspected := suspected ∪ {p};

upon event ⟨ ◇P, Restore | p ⟩ do
    suspected := suspected \ {p};

upon leader ≠ maxrank(Π \ suspected) do
    leader := maxrank(Π \ suspected);
    trigger ⟨ Ω, Trust | leader ⟩;
```

fig. 3.5. "Perfect" leader election algorithm (left) and "eventual" leader election algorithm (right).

The most important property of the *perfect* algorithm is **LE2** (as defined in Section 1), which ensures that once a leader is elected, it is guaranteed that all previously suspected leaders have indeed crashed (fig. 3.3). Although leader election does not occur synchronously due to the asynchronous nature of message delivery, the strong guarantees of *P* ensure that no two correct processes will simultaneously consider different leaders alive—effectively preventing *split-brain* scenarios in the perfect algorithm.

THE PROBLEM

The goal is to determine how to select the appropriate failure model and leader election algorithm for the mission scenario. Specifically, we aim to answer the following research question: *Which of the two algorithms—the perfect or eventual leader election—yields the most efficient mission KPIs?*

Upon closer examination, the *eventual* leader election algorithm can be viewed as a superset of the *perfect* algorithm when configured with the same base timeout as the latter. Furthermore, it accommodates systems that may exhibit asynchronous behavior. In practical deployments, if the use case tolerates temporary inconsistencies or split-brain scenarios, the *eventual* algorithm may be the more resilient and realistic choice. However, rather than making assumptions about whether split-brain behavior is acceptable in our scenario, we abstract away such concerns and evaluate the algorithms based solely on mission-level KPIs. This allows us to assess system quality and effectiveness from a purely outcome-driven perspective.

In practice, both leader election algorithms can result in a split-brain state under certain conditions. For example, if the system is incorrectly assumed to be synchronous while a temporary network partition occurs, the *perfect* algorithm may mark some UAVs as crashed for the remainder of the mission. These UAVs will then be excluded from future task assignments, even though they continue to operate—potentially monitoring previously assigned areas. This misclassification can lead to reduced KPIs due to redundant task execution and underutilization of available resources.

The figure 3.6 summarizes the strengths and limitations of each algorithm. The blue border highlights the focus of this research: evaluating how the *eventual* model impacts mission KPIs, and whether it is possible for it to outperform the *perfect* model under conditions where a longer timeout (adapted to observed delays) mitigates false crash detection.

Lower timeout	Larger timeout	Dynamic timeout
Fast response	Always one leader	Faster response
Multiple leaders	Decision delays	Eventually one leader
Double coverage		Temporary inconsistency

fig. 3.6. Pros and cons of the different timeout strategies.

In the case of the *eventual* algorithm, a split-brain state is considered an expected condition, and the mission logic must be designed accordingly—specifically, with mechanisms to manage UAVs marked as *suspected*. This section introduces the main hypotheses guiding the experimental evaluation:

1. In scenarios where no UAV crashes occur, KPI performance is primarily influenced by the extent of the split-brain state. If the network behaves synchronously and an optimal timeout can be theoretically derived, the *perfect* leader election algorithm is expected to yield better KPIs. In contrast, the *eventual* algorithm may incur KPI losses while attempting to determine a sufficient retry count to converge.
2. In scenarios with expected UAV crashes and a suboptimal timeout configuration for the *perfect* algorithm, KPI degradation is anticipated. Longer delays in leader reelection may result in periods where areas are left unmonitored or underserviced.
    1. As the number of UAV crashes increases, overall KPI performance is expected to decline further due to the cumulative impact of increased delays and prolonged underservice intervals.
3. Beyond a certain point, the *eventual* algorithm is expected to outperform the *perfect* algorithm with a suboptimal timeout, as the negative impact of delayed recovery in *perfect* will outweigh the cost of temporary split-brain resolution in *eventual*.

3.2. Benchmark Experiments

The simulation environment includes a wide range of parameters, such as perimeter shape and size, UAV altitude (which determines surveillance diameter), UAV speed, threat spawn rate, UAV count, and network-related settings. However, due to the real-time nature of the simulation—where a 3-minute simulation requires 3 minutes of actual execution time—it is not feasible to explore all possible parameter combinations within a reasonable timeframe. As a result, this research focuses on varying only a selected subset of parameters: network configuration, timeout settings, and UAV crash rates. The remaining parameters are held constant across all experiments, with their values summarized in Table 3.1.

All parameters in Table 3.1, with the exception of UAV count, can be described in a straightforward manner. The perimeter length is set to 1200 meters for two main reasons. First, the ability to visually debug specific simulation runs is constrained by the limited resolution of the console-based terminal interface. Second, a 1200-meter perimeter likely represents the minimal size of a strategic asset, such as a factory.

The surveillance (video) diameter is fixed at 30 meters, calculated using (3.2), where the field of view (FOV) is 80∘ and the UAV altitude h is 20 meters. A UAV speed of 12 m/s corresponds to the average cruising speed of the DJI Matrice 300 RTK, making it a realistic parameter for real-world deployments.

Table 3.1

Baseline simulation parameters

Parameter name	Unit	Value
Perimeter height	Meters	500
Perimeter width	Meters	100
UAV video diameter	Meters	30
UAV speed	Meters/sec	12
UAV count	UAV	10
One simulation time	Minutes	3
Threat rate	Threat/min	5
Threat mean lifetime	Seconds	30
Simulation time	Seconds	300
Delta timeout	ms	3000

The simulation time is set to 5 minutes, representing a balance between simulating a realistic mission duration and the need to conduct a statistically meaningful number of simulation runs. Although the selected UAV's battery allows for significantly longer flight times (up to 10 times the simulated duration), a 5-minute simulation enables the execution of approximately 120 runs within a 12-hour evaluation window.

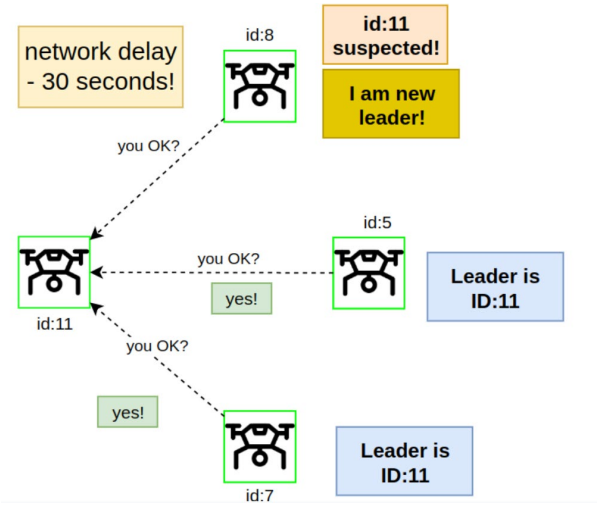

fig. 3.7. Example of the eventual leader election algorithm execution. Due to partial-synchrony, nodes are in temporary disagreement on who is the leader.

To estimate the required number of simulation runs necessary to obtain a reliable average for key performance indicators (KPIs), we use the following method:

$$n = \left(\frac{z \cdot \sigma}{E}\right)^2, \tag{3.1}$$

where Z – Z-score, which represents confidence level (e.g. 1.96 for 95%);

n – minimum number of simulations needed;
σ – standard deviation of the data set;
E – error margin.

For a 95% confidence level, given the observed standard deviation and a specified margin of error 1, it was determined that 56 simulation runs are required to ensure that the sample mean with σ = 3.82. Based on the 5-minute simulation duration, this allows for approximately 2 distinct parameter configurations (scenarios) to be evaluated within a 12-hour testing window.

$$h = \frac{D}{2 \times \tan\left(\frac{F}{2}\right)}, \tag{3.2}$$

Where F – field of view of the camera, deg;
h – UAV flight height, m;
D – meters of ground captured on video, m.

NETWORK MODEL

To simulate an unstable wireless network environment, a log-normal distribution was chosen to model message delays. Unlike a standard normal distribution, the log-normal distribution allows for occasional longer delays, better reflecting real-world wireless communication irregularities. The distribution is configured with parameters μ=6 and σ=1.5, introducing variability in delivery times. Additionally, a 1% UDP packet drop rate is applied to further emulate unreliable network conditions.

To improve the reliability of heartbeat acknowledgments under conditions of high-variance delay and packet loss, a retry mechanism is introduced. In this setup, node A periodically sends heartbeat messages to node B and awaits an acknowledgment (ACK). The one-way network delay D is modeled as a log-normal random variable, $D \sim LogNormal(\mu,\sigma)$, with parameters μ=6 and σ=1.5. This results in heavy-tailed delay behavior, where extreme delays—occasionally exceeding 100 seconds—are statistically plausible. In addition, the communication channel is configured with an independent packet drop probability p_{drop}=0.01.

In the naïve implementation, node A would mark node B as failed if an ACK is not received within a timeout threshold X, potentially leading to false failure detections due to packet loss or rare but valid delay outliers. To mitigate this, the retry mechanism is introduced: if no ACK is received within time X, node A resends the heartbeat, up to n times. Each retry is modeled as an independent sample from the same log-normal delay distribution and is independently subject to packet loss.

The probability that a single retry attempt fails—either due to packet drop or delay exceeding X—is given by equation (3.3). The cumulative distribution function (CDF) of the log-normal delay is equation (3.4). Assuming independence, the probability that **all n retries** fail is equation (3.5) (fig. 3.8).

$$P_{\text{fail}}(X) = p_{\text{drop}} + (1 - p_{\text{drop}}) \cdot \mathbb{P}(D > X) \tag{3.3}$$

$$\mathbb{P}(D \leq X) = \Phi\left(\frac{\ln(X) - \mu}{\sigma}\right) \quad \Rightarrow \quad \mathbb{P}(D > X) = 1 - \Phi\left(\frac{\ln(X) - \mu}{\sigma}\right) \tag{3.4}$$

$$P_{\text{total fail}}(X, n) = [p_{\text{drop}} + (1 - p_{\text{drop}}) \cdot \mathbb{P}(D > X)]^n \tag{3.5}$$

where Φ is CDF. By choosing a moderate timeout value X (e.g., 4000 ms) and a limited number of retries (e.g., n=5), the failure probability can be reduced to below 10^{-8}, effectively ensuring p99.999999 reliability, even under high delay variance and 1% packet drop. This transforms the tail-heavy log-normal delay distribution into a more favorable minimum-of-n distribution, thus allowing fault detection mechanisms to be both responsive and robust.

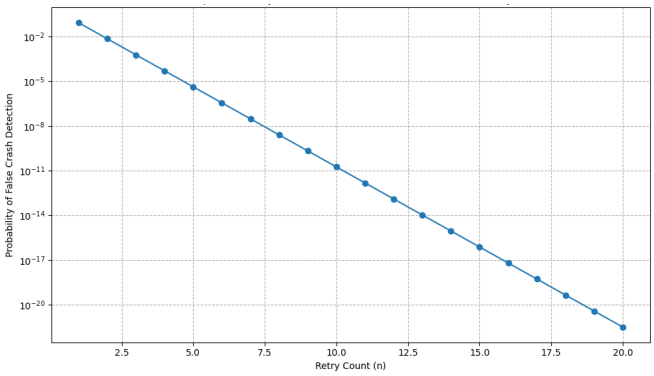

fig. 3.8. Probability of the false crash detection depending on retry amount.

The theoretical model shows that 4 retries with a delay threshold D=3500 ms yield a 99.999% probability that no false crash detection will occur. More than 200 simulation runs, each 5 minutes long, confirm that with a perfect timeout of 15 seconds and a retry interval of 3500 ms, no false crash detections were observed.

HOW UAV COUNT AFFECTS KPI?

Experimental results showed that both the *lifetime monitor* KPI and the *threat registration time* (t. reg.) KPI exhibit a logarithmic dependency on the number of UAVs. Given a 1200-meter perimeter, deploying 10 UAVs implies that each UAV is responsible for monitoring approximately 100 meters. Increasing the number of UAVs beyond 10 provides only marginal improvements in performance while significantly increasing network traffic.

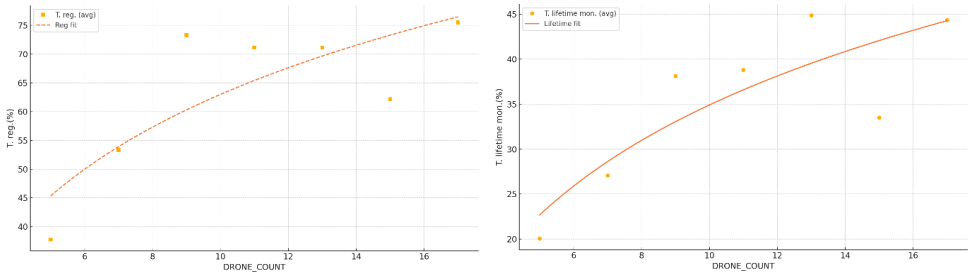

fig. 3.9. Approximated function based on simulation data of KPI dependency on drone count (left – T.Reg, right – T. lifetime mon.)

IDEAL NETWORK VS POOR

First, an ideal network experiment was conducted with a packet drop rate of 0 and no delay. Subsequently, the degraded network model described in the previous section was applied. As shown in the table below, the unreliable network conditions can lead to a KPI reduction of up to 7.2% (Table 3.2).

Table 3.2

Comparison of "ideal" network and baseline parameters ("bad" network).

KPI	"Good" network	"Bad" network
T. reg.(%)	71	71
T. contr.(%)	52	46
T. lifetime mon.(%)	64	60
T. lifetime contr.(%)	47	40
Sum	**234**	**217**

3.3. Analysis of eventual algorithm convergence

In this chapter, we analyze the split-brain state—examining the conditions under which it may occur, its expected duration based on system parameters, and its impact on mission KPIs.

SPLIT-BRAIN

Not all false crash detection events lead directly to a split-brain state. For example, if a false crash is detected between two UAVs with low IDs—neither of which is the current leader—the system's functionality remains unaffected. Both UAVs will continue to receive commands from the current leader and send event updates to it. However, problems arise when a false crash is detected between a UAV with ID equal to *Current Leader ID – 1* and the current leader itself. In this case, the UAV may elect itself as the new leader and:

- cease sending events to the original leader, and
- begin processing commands from both itself and the original leader.

The specifics of how such conflicting command handling is implemented may vary, but these optimizations fall outside the scope of this research.

Similar split-brain scenarios can occur when a false crash is detected between the current leader and any other UAV, potentially leading to inconsistent system behavior.

HOW FAST EVENTUAL TIMEOUT CONVERGE?

In this set of experiments, the goal was to determine the point in time after which no further timeout adjustments occur — i.e., when all UAVs have stabilized and no false crash detections persist. As expected, this strongly depends on the network model. The experiments were conducted using the previously described degraded network model.

The results demonstrate that the *delta timeout* (DT) parameter significantly influences convergence time. However, increasing DT comes at a cost—resulting in a less responsive final timeout configuration. Experiments showed that even with a large DT of 3 seconds, and UAV group sizes greater than 5, false positives can persist for up to 10 minutes. Nevertheless, most false crash detections occur within the first 3 minutes.

When DT is reduced to 1 second, even for a small group of 3 UAVs, false positives persist for nearly 9 minutes and are uniformly distributed over time. A DT of 2 seconds produces similar results to $DT = 1$ second across all tested UAV counts, suggesting diminishing returns in convergence efficiency with smaller DT values.

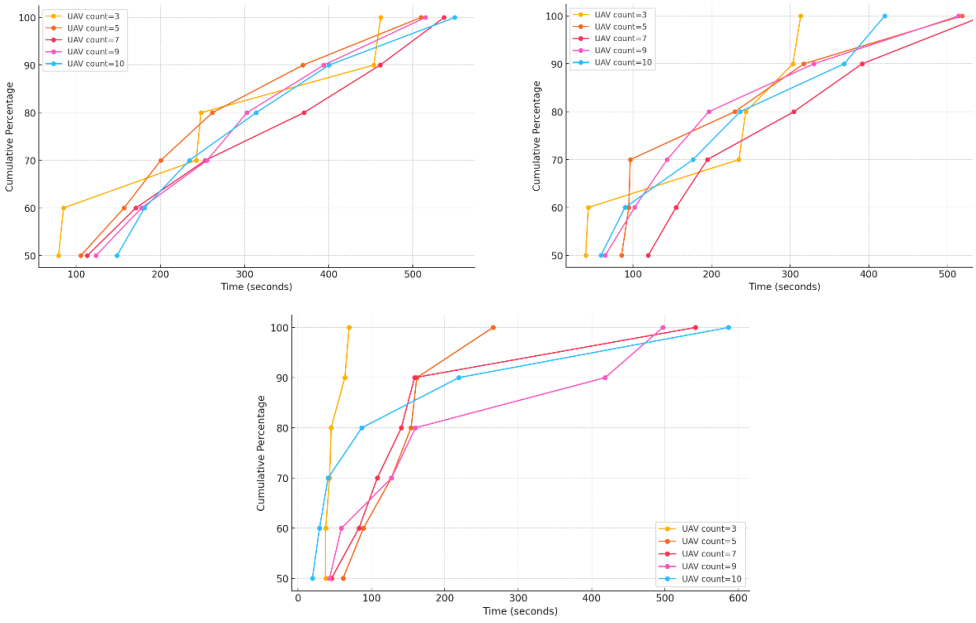

fig. 3.10. Cumulative amount of false crashes during simulation depending. Delta timeout = 1s (left), 2s (right), 3s (center).

HOW EVENTUAL ALGORITHM AFFECTS KPI?

In the previous chapter, we observed that under degraded network conditions—even with an aggressive delta timeout of 3 seconds—false positives can persist for an extended period. Given that the total simulation time for all experiments is limited to 5 minutes, false crash detections often span over 2/3 of the duration. However, the split-brain analysis also showed that not all false positives have a direct impact on mission execution. To quantify the overall effect, we conducted experiments under bad network conditions, without any actual UAV crashes, and compared the performance of the *eventually perfect* algorithm against the *perfect* one.

As shown in the table above, the *eventual* algorithm results in a 4% KPI loss due to the split-brain state. When the delta timeout is set to 1000 ms, the KPI loss increases to 6.5% (Table 3.3).

3.4. Crash scenarios analysis

In this chapter, we analyze the impact of longer timeout values in scenarios involving UAV crashes, with a focus on how increasing crash rates influence system behavior. The final hypothesis evaluates whether the *eventual* algorithm can demonstrate an advantage over the *perfect* algorithm configured with a conservative timeout.

Table 3.3

No UAV crashes. How "eventual" algorithm affects KPI.

KPI	Perfect	Eventual (dt=3s)	Eventual (dt=1s)
T. reg.(%)	71	68	69
T. contr.(%)	46	43	41
T. lifetime mon.(%)	60	57	55
T. lifetime contr.(%)	40	40	38
Sum	**217**	**208**	**203**

HOW LONGER TIMEOUTS AFFECTS KPI IN CRASH SCENARIOS?

The table below compares the performance of the *perfect* algorithm across various timeout values, ranging from the ideal 15 seconds up to 180 seconds. This comparison helps to estimate the maximum potential gain achievable by using the *eventual* algorithm, assuming it can converge to the ideal timeout over time.

With a 30% crash rate, the conservative timeout configuration results in a 10.8% KPI loss compared to the ideal timeout. At a 50% crash rate, the loss is slightly lower at 8.8%. However, at a 70% crash rate, the simulations indicate the highest degradation, with a KPI loss of 17%.

A comparison between the ideal timeout and the large enough timeout (180 seconds) reveals a different trend. With a 30% crash rate, the KPI loss reaches 18%; at 50% crashes, the loss increases to 20%; and at 70%, the loss slightly decreases to 19%.

CAN EVENTUAL OUTPERFORM PERFECT?

In the final experiment, we evaluate how all three configurations perform under varying crash rates and degraded network conditions. With a 30% crash rate, the *eventual* algorithm exhibits only a 2% KPI loss compared to the ideal timeout. It is worth noting that some simulations resulted in a timeout shorter than the ideal 15 seconds. This behavior is expected and may partly explain the minimal difference observed between the two algorithms. At a 50% crash rate, the largest discrepancy occurs, with the *eventual* algorithm showing a 3.7% loss relative to the ideal. At a 70% crash rate, the difference becomes negligible, with less than 1% deviation between *eventual* and *ideal* configurations.

3.5. Conclusions

The original goal of this research was to answer the question: *Can eventually consistent distributed algorithms be used in cooperative UAV missions with shared state?* During the development of the experimental testbed, this question was progressively refined. The selected mission scenario for cooperative UAV coordination was perimeter patrol. To enable meaningful comparison between algorithm types, a set of KPIs was defined to quantify the impact of each algorithm on mission success.

Among various mission control architectures, a leader-based approach was chosen. For the distributed coordination mechanism, a monarchical leader election algorithm [1] was implemented. As the research evolved, the central question was reformulated more precisely as: *Can eventual leader election outperform perfect leader election in the context of expected UAV crashes?* This refined question served as the foundation for the formal hypotheses presented in 3.1.

Throughout the research, several key findings and observations were made:

- The core UAV control logic has a significantly greater impact on mission performance than potential inconsistencies caused by distributed coordination. Therefore, before evaluating the effects of disagreement, it is essential to first identify the most effective control algorithm—even for relatively simple 2D missions.
- UAV patrol behavior, in general, results in suboptimal KPI values, even when using an unrealistically high number of UAVs (e.g., 120 meters of perimeter per UAV).
- In addition to modeling network delays and packet drops, it is also important to simulate temporary network partitions to capture more realistic failure scenarios.
- Real-time simulation is valuable for uncovering race conditions and other timing-related issues; however, it is time-consuming and limits the number of runs needed to achieve statistically reliable results.
- Advanced edge-case scenarios can still trigger subtle bugs. For example, bugs may occur when UAVs report events out of order, and the system does not properly associate them with the correct sender or event sequence. (Note: this issue should be documented and resolved in future work.)

- Even if consensus is not reached in *eventual* mode, system behavior can be improved by implementing logic to handle situations involving multiple leaders.
- KPI performance follows a logarithmic relationship with the number of UAVs.
- High network delays and packet drops can degrade KPIs by up to 7%.

In the no-crash experiments, Hypothesis 1 was confirmed. The *eventual* algorithm with a delta timeout of 3 seconds showed a 4% KPI loss compared to the ideal timeout configuration. Under the same conditions, a delta timeout of 1 second resulted in a 6.5% loss. These results demonstrate that, although convergence time for a group of 10 UAVs does not significantly depend on the delta value, the fact that convergence follows a logarithmic trend—and that most false crash detections occur early—translates into measurable KPI gains when using a larger delta.

Hypothesis 2.1 was confirmed by the experimental results. The simulations showed the greatest KPI loss—17%—in scenarios where the majority of UAVs crashed during the mission. However, the scenarios with 30% and 50% crash rates exhibited nearly identical losses, with only a 2% difference, and the 50% case even showed a slightly smaller loss than the 30% case. This indicates that the cost of longer delays does not depend solely on the number of crashes and is not strictly linear with respect to the crash rate.

Hypothesis 3 was confirmed by the experimental results. The *eventual* algorithm consistently outperformed the *perfect* algorithm across all scenarios, despite starting at a disadvantage. The impact of convergence time on KPI was shown to depend on the crash rate. The largest KPI loss relative to the ideal timeout (3.7%) occurred in the 50% crash scenario. This was followed by a 2% loss in the 30% crash scenario, and less than 1% in the 70% crash scenario.

This research demonstrated that eventually consistent distributed algorithms—specifically, eventual leader election—can be effectively applied in cooperative UAV missions such as perimeter patrol. Although the core control logic of UAVs has a greater influence on mission success than occasional inconsistencies, the *eventual* approach exhibited strong resilience in crash-prone environments and, under certain conditions, even outperformed the *perfect* leader election strategy. Key findings revealed that mission performance is influenced by network instability, UAV count, and convergence time in non-linear ways. Additionally, incorporating logic to handle inconsistencies—such as the presence of multiple leaders—can further enhance system robustness. Overall, *eventual* algorithms offer a practical, fault-tolerant alternative to idealized coordination models, particularly in real-world scenarios where crashes and network disruptions are inevitable.

4. DYNAMIC CHARACTERISTICS OF COOPERATIVE UAV NETWORK

Two types of AI are involved: onboard UAV AI and a more powerful ground-based central AI node. This research focuses on analyzing the threat detection aspect of the patrol mission. In this chapter, the UAV patrol mission is modeled as a Gordon–Newell Queuing Network to analyze the dynamic characteristics of the system, identify bottlenecks, model how AI performance impacts mission KPIs, and compare the effects of improving the central AI node versus the onboard AI.

4.1. Introduction

In the previous chapter, simulation was used to analyze how different leader election configurations affect mission KPIs. Threats in the simulation were modeled as 2D ASCII symbols, and the detection process was relatively simple—the 30×30 character array was passed to the UAV's threat detection module, and if a threat symbol was found, the threat was considered detected. While this approach poorly approximates real-world conditions, it was sufficient for the purposes of that study. In real-world scenarios, UAVs would be equipped with video cameras and would capture numerous objects during patrols. Most of these objects — whether static or dynamic—would not constitute real threats. To enable automatic threat detection, deep learning models can be used. The specifics of such models lie beyond the scope of this research. Instead, based on existing literature in the field [43] [44], assumptions regarding AI accuracy, hardware requirements, and performance will be adopted. The AI service will be treated as a black box.

THE PROBLEM

This chapter aims to address several key research questions related to the performance of the UAV patrol system. First, it investigates the structure of the system to identify its main components and potential performance bottlenecks. Second, it seeks to determine the theoretical limits of system throughput and to analyze the response time distribution using queueing theory. Finally, the study explores which aspects of the system should be optimized— such as onboard AI accuracy or central AI response time —in order to achieve the best possible mission KPIs.

4.2. The model

To select an appropriate model for answering the defined research questions, the overall process must first be described. A number of UAVs continuously monitor a designated area, each moving from the location of its current task point A to the point B. During patrol, objects captured on the UAVs' onboard cameras are classified in real time using the onboard AI module. However, onboard systems are resource-constrained, and their classification accuracy is

limited. Whenever the onboard AI detects a potential threat, it sends a video segment to the central AI node for a second opinion. In most cases, the central AI is expected to classify the object with sufficient confidence and returns a decision to the UAV, which then either disregards the object or initiates tracking. In rare cases where the central AI has low confidence in its classification, the video is forwarded to a human operator for further analysis. Once the human makes a decision, a command is sent back to the UAV to determine its response. The area is patrolled by N UAVs, and it is assumed that each UAV generates central AI requests with some average frequency. If an object is confirmed to be a real threat, the UAV begins tracking it.

Each component in this system can be represented as a queue, and the entire system can be modeled as a queuing network. The network handles a single type of job: a request from a UAV to determine whether a detected object constitutes a real threat. At this stage, the focus shifts to understanding the job arrival rate. Queuing networks are generally categorized as either open or closed. Closed networks are typically used to model systems such as computer architectures or manufacturing loops, where a fixed number of jobs or customers circulate continuously. In contrast, open networks are suitable for modeling systems with external arrivals that follow a stochastic process, such as call centers, web services, or retail checkouts. In the current scenario, the number of UAVs is fixed, and they behave like users interacting with a shared service, issuing requests at average intervals. Therefore, the system is best represented as a closed queuing network, as illustrated in the figure 4.1. Gordon–Newell Queuing Networks (GNQNs), named after their inventors, represent a class of closed Markovian queuing networks, where all service times follow a negative exponential distribution. In this model, we work with M/M/1 queues that are interconnected based on predefined routing probabilities. The average service time at queue *i* is given by $\mathbb{E}[S_i] = 1/\mu_i$.

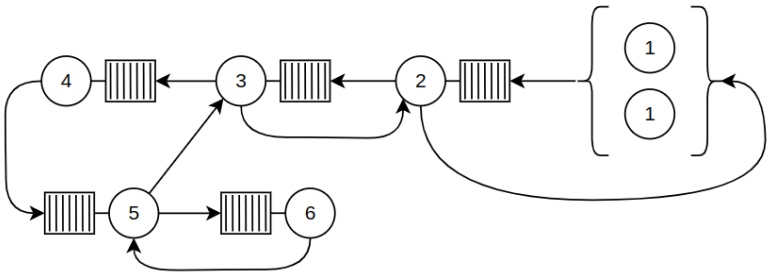

fig. 4.1. Closed Gordon-Newell QN model for cooperative UAV network.

Node 1 represents K UAVs that periodically requests job processing. It is a special infinite server node, which will be described in detail later. The interval between requests has a mean value typically denoted in the literature as *E[Z]*. As K increases—that is, as more UAVs are added to the network—the system load logically increases, since more UAVs imply more potential tasks to process. Node 2 represents the request transmission over a cellular network. In this stage, a high-quality video chunk is streamed to the central server via a 4G/5G network. The service time for this node is denoted as $E[S_{CellularTransmission}]$.

The next stage, Node 3, is the central commander. This could be a server room within the facility, or a mobile container equipped with the necessary hardware. This node handles CPU-based computations such as task distribution and mission control logic, with a service time $E[S_{CommanderRequest}]$. Node 4 represents the central AI processing unit, which performs intensive computations using deep learning models to detect threats in the video streams. Its service time is $E[S_{CentralAI}]$. Node 5 models the network connection from the central commander to the human operator, likely through a WAN gateway, with service time $E[S_{WANTransmission}]$. Finally, Node 6 represents the human operator, who views the video and makes decisions about the next actions. This node has a service time of $E[S_{OperatorProcessing}]$.

Table 4.1

Response times and thruput of QN components by K

K	$E[R_1(K)]$	$E[R_2(K)]$	$E[R_3(K)]$	$E[R_4(K)]$	$E[R_5(K)]$	$E[R_6(K)]$	X(K)
1	20	0.03	0.20	1.50	0.00	0.03	0.05
4	20	0.03	0.21	1.85	0.00	0.03	0.18
7	20	0.03	0.21	2.37	0.00	0.03	0.31
10	20	0.03	0.22	3.17	0.00	0.03	0.43
13	20	0.03	0.22	4.43	0.00	0.03	0.53
16	20	0.03	0.23	6.40	0.00	0.03	0.60
19	20	0.03	0.23	9.28	0.00	0.03	0.64
22	20	0.03	0.23	13.02	0.00	0.03	0.66
25	20	0.03	0.23	17.27	0.00	0.03	0.67
28	20	0.03	0.23	21.72	0.00	0.03	0.67

AI SERVICE RATE ESTIMATION

Based on recent research, a reliable AI-based threat classification system using UAV video can operate on 1.5-second 4K clips, which are sufficient for accurately identifying threats such as drones, humans, or vehicles in suburban environments [43] [44]. A 4K video at 30 FPS produces about 45 frames per classification cycle; when compressed with H.264, the average size per clip is approximately 12–20 MB, depending on scene complexity. In the study by Baykara et al [43], a UAV-based real-time object detection and classification system was developed, achieving classification of moving objects (humans, vehicles, or faulty detections) using a SqueezeNet CNN model, with each frame classified independently, and maintaining classification for up to 1.5 seconds based on confidence. The system processed video frames sequentially, requiring approximately 6.75 seconds to fully classify a 1.5-second 1080p video (45 frames) on a low-end CASPER laptop with an Intel Core i5-3210M CPU, 16 GB RAM, and NVIDIA GTX 950M GPU, which cost around $80–$90 at the time. Extrapolating to a modern setup, a real-time API-based classification system capable of processing 1.5 seconds of 4K video in under 1 second would require an NVIDIA RTX 4090 or A100 GPU, a high-end CPU (e.g., AMD Ryzen 9 7900X), 64 GB DDR5 RAM, and NVMe SSD storage, with a total

cost of approximately $5,000-$10,000, enabling precise threat detection and classification in high-resolution UAV feeds.

Table 4.2

Utilization and expected queue length of QN components by K

K	$\rho_1(K)$	$\rho_2(K)$	$\rho_3(K)$	$\rho_4(K)$	$\rho_5(K)$	$\rho\_6(K)$	$E[N_1(K)]$	$E[N_2(K)]$	$E[N_3(K)]$	$E[N_4(K)]$	$E[N_5(K)]$	$E[N_6(K)]$
1	0	0.00	0.01	0.07	0.00	0.00	0.92	0.00	0.01	0.07	0.00	0.00
4	0	0.01	0.04	0.27	0.00	0.01	3.62	0.01	0.04	0.33	0.00	0.01
7	0	0.01	0.06	0.46	0.00	0.01	6.18	0.01	0.07	0.73	0.00	0.01
10	0	0.01	0.09	0.64	0.00	0.01	8.53	0.01	0.09	1.35	0.00	0.01
13	0	0.02	0.11	0.79	0.00	0.02	10.52	0.02	0.12	2.33	0.00	0.02
16	0	0.02	0.12	0.90	0.00	0.02	11.99	0.02	0.14	3.84	0.00	0.02
19	0	0.02	0.13	0.96	0.00	0.02	12.85	0.02	0.15	5.96	0.00	0.02
22	0	0.02	0.13	0.99	0.00	0.02	13.21	0.02	0.15	8.60	0.00	0.02
25	0	0.02	0.13	1.00	0.00	0.02	13.31	0.02	0.15	11.49	0.00	0.02
28	0	0.02	0.13	1.00	0.00	0.02	13.33	0.02	0.15	14.47	0.00	0.02

BUZEN'S ALGORITHM

We will use Buzen's convolution algorithm to calculate various metrics of interest for this module. Buzen's convolution algorithm is an efficient method for solving the normalization constant *G(K)* in closed queueing networks with a finite number of customers *K*. The key idea is to recursively compute *G(K)* using a dynamic programming approach, avoiding the direct summation over all possible states. Given service rates D_i for each queue *i* and the number of customers *K*, the normalization constant is computed as:

$$G(M,K) = \sum_{\mathbf{n} \in \mathcal{I}(M,K)} \prod_{i=1}^{M} D_i^{n_i} \tag{4.1}$$

where:

- *G(M,K)* is the normalization constant for a closed queuing network with M queues and K total customers.
- $\mathcal{I}(M,K)$ denotes the set of all possible state vectors $n = (n_1, n_2, \ldots, n_M)$.
- D_i represents the service demand at queue *i*.
- n_i is the number of customers in queue *i* in state *n*.
- The product $\prod_{i=1}^{M} D_i^{n_i}$ evaluates the contribution of each state *n* to the normalization constant.

Buzen reformulated this summation as an iterative recurrence relation:

$$G(M,K) = G(M-1,K) + D_M\, G(M,K-1) \tag{4.2}$$

To start this recursion, we need boundary values. These can be derived as follows. When there is only 1 queue, by definition, $G(1,k) = D_1^k, \quad \text{for all } k \in \mathbb{N}$. Also, by the fact that there is only one way of distributing 0 customers over M queues, we have $G(m,0) = 1, \quad \text{for all } m$.

ASSUMING ONE INFINITE SERVER NODE

A special case arises when analyzing a QN that includes a single infinite-server station. In such a network with *K* customers, the infinite-server station can be treated as a K-server station (node 1 in our case). It is reasonable to assume that the network contains only one infinite-server station; if multiple exist, they can be merged into a single equivalent station. The primary modification to the computational scheme involves adjusting the initialization to account for the presence of the infinite-server node. The steady-state distribution of the number of customers in this network, given a total of K customers, is provided in [45]:

$$\Pr\{N = n\} = \frac{1}{G(M,K)} \frac{D_1^{n_1}}{n_1!} \prod_{i=2}^{M} D_i^{n_i} \tag{4.3}$$

$$G(m,0) = 1, \quad \text{for } m = 1,\ldots,M, \tag{4.4}$$

$$G(1,k) = \frac{D_1^k}{k!}, \quad \text{for } k = 0,\ldots,K. \tag{4.5}$$

The only irregularity in the queueing network appears in the computational scheme at initialization, while the remaining computations remain unchanged (4.5).

MEAN VALUE ANALYSIS

For the reconciliation of the results achieved by Buzen's algorithm, the mean value analysis approach can be used [45]. For infinite-server nodes, there is no waiting time, so the response time equals the service time:

$$\begin{gathered} E[R_j(K)] = E[S_j], \quad E[\hat{R}_j(K)] = D_j \\ \mathbb{E}[\hat{R}_i(K)] = (\mathbb{E}[N_i(K-1)] + 1)\, D_i \\ E[N_i(K)] = X(K) E[\hat{R}_i(K)] \\ \sum_{i=1}^{M} E[N_i(K)] = K, \quad X(K) = \frac{K}{E[\hat{R}(K)]} \end{gathered} \tag{4.6}$$

The expected response time, number of customers and throughput (equation (4.6)). These equations extend MVA to IS nodes, maintaining analytical consistency.

SYSTEM PARAMETERS

To calculate basic values like throughput, response time, and utilization, we need service demands D_i *(Di=Vi·E[Si])*. First, the job itself, as mentioned in the previous chapter, will be a 4K video chunk of 2 s length. We assume a standard streaming platform bitrate of 15 Mbps, so a total of 30 Mb, or roughly a 4 MB file per UAV request. For simplicity, we do not count

overhead from header information or retransmissions. We start by estimating service times for each node. The objective is to determine indicative benchmark values; therefore, the focus is on obtaining estimates that are as representative as possible. While each node's service time could be the subject of dedicated research, this study will rely on existing literature, aiming to derive an average value for each parameter. Node 1 simply depends on what will be the rate of objects that needs to be classified. So, in theory, there will be the first layer on the onboard setup that can identify any unregular object in the landscape. For example, if the rate of such objects is 10 per minute, then $E[Z]=6$. Node 2, the cellular network transmission, we assume 1 Gbps of real bandwidth available. So, $E[S_{CellularTransmission}]=30/1000=0.03s$. For node 3, the central commander API, the typical routine would be to rebuild mission state using new information from the request, run multiple intensive calculations to understand the optimal strategy for which UAV should monitor which area, and which UAV should monitor the threat. We estimate service time as $E[S_{CommanderRequest}]=0.2$. For node 4, the central AI, we start with $E[S_{CentralAI}]=1.5s$. Node 5 is the transmission over WAN, because the operator center is probably located far from the security object. For WAN delay plus transmission we roughly estimate $E[S_{WANTransmission}]=0.3$. For human operator analysis, we set $E[S_{OperatorProcessing}]=30s$. All QN parameters are summarized in the table below (Table 4.3).

Table 4.3

QN model parameters

Node	Name	E[S] (s)	V_i	D_i
1	Interval time between UAV requests (depends on object rate) (E[Z])	20	1	20 (3 obj/min)
2	Cellular transmission ($E[S_{CellularTransmission}]$)	0.03	1	0.03
3	Central Commander API ($E[S_{CommanderRequest}]$)	0.2	1	0.2
4	Central AI ($E[S_{CentralAI}]$)	1.5	1	1.5
5	WAN Transmission ($E[S_{WANTransmission}]$)	0.3	0.001	0.0003
6	Human operator analysis ($E[S_{OperatorProcessing}]$)	30	0.001	0.03

4.3. Performance depending on UAV count

In Table 4.1 and Table 4.2, we see all possible performance values calculated using Buzen's algorithm, for different UAV count (K) and for object rate of 3/min. The most interesting component is the system's bottleneck which is the component with largest service time – Node 4 (central AI). The maximum throughput of the system reached around 28 UAVs. The most important value is system's response time, utilization and average number of nodes in queue.

For reference K of 10 UAVs, bottleneck node 4, we have 64%, 1.35 average queue and response time of 3.17s. On below figure (fig. 4.2), we can see how fast response time grows depending on UAV count and different mean request interval.

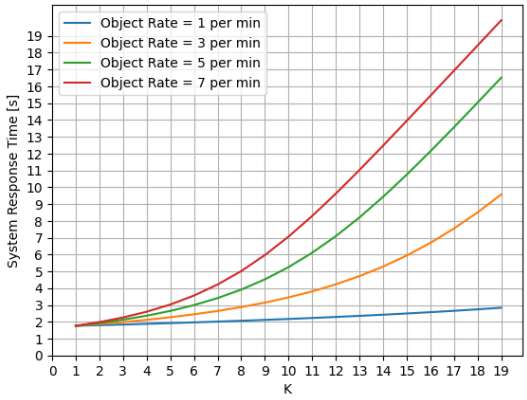

fig. 4.2. System's response time depending on K (UAV count). Family of function by identified object rate.

The performance evaluation shows that as UAV count (*K*) increases, system response time grows non-linearly, especially beyond K=10. The central AI (Node 4) is the system's bottleneck, reaching near-full utilization (up to 99.6%) at high UAV counts. This causes sharp increases in response time and queue size. At K=10, the system operates at 64% utilization with an average response time of 3.17s, indicating an efficient working point. However, higher object rates significantly degrade performance, and the system saturates beyond K=13. Therefore, the maximum sustainable throughput is achieved around K=10–13 UAVs.

RESPONSE TIME DISTRIBUTION

Often, the average response time is not an adequate metric for non-functional system requirements, as it does not reflect the quality perceived by users. Instead, it is more informative to assess the time range within which 95% of responses occur. Thanks to Buzen's algorithm, we are able to calculate the probability of each system state. With both the expected response time per state and the corresponding state probabilities available, it becomes possible to construct the probability distribution function (PDF) of the system's response time (fig. 4.5). The algorithm for constructing such a PDF is presented below (fig. 4.4). This provides a more realistic view of the system's performance. For instance, even with a relatively small number of UAVs, the guaranteed response time may reach up to 6 seconds, which is already quite slow (fig. 4.3).

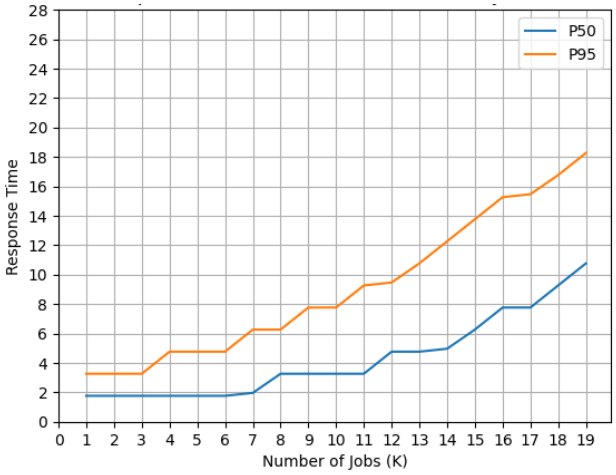

fig. 4.3. System's response time depending on K by percentiles 90 and 95.

The goal was to create a function that constructs a discrete probability distribution dictionary based on the provided parameters—a vector of service demands and a specified value of K. Our objective is to understand the system's response time from the user's perspective. For example, if the parameters are $D = [1,1,1,1]$ and $K=1$, then we obtain a 75% probability that the server will respond in 3 seconds and a 25% probability of a 4-second response. The algorithm for computing this distribution is presented below.

We begin by generating all possible states of the queueing network. For each state, we calculate the response time by iterating through each station and computing its contribution based on the number of jobs in its queue in that state. It is crucial to add one job unconditionally to each queue during this calculation; otherwise, in the simple example above, we would incorrectly obtain a 25% chance that the server's response time is zero. In the final step, we compute the probability of each state using the normalization constant.

The precision of the resulting response time distribution depends heavily on the value of K. As K increases, there are more possible system states, which leads to a greater variety of potential response times. Consequently, for lower values of K, the distribution plots may not appear representative. To improve interpretability, we group (bin) many low-probability response times into 1-second intervals, resulting in more informative and readable plots.

Require: Service demand vector $D = (D_1, D_2, \ldots, D_M)$, population K
Ensure: Response time probability distribution $\mathcal{P}$
1: Compute normalization constants $G \leftarrow \text{Buzen}(K, D)$
2: Initialize empty distribution $\mathcal{P}$
3: **for** each state $S = (s_1, s_2, \ldots, s_M)$ such that $\sum s_i = K$ **do**
4: Compute response time:

$$R \leftarrow \sum_{i=2}^{M} (s_i + 1) D_i$$

5: Compute state probability:

$$P(S) \leftarrow \frac{D_1^{s_1} / s_1! \cdot \prod_{i=2}^{M} D_i^{s_i}}{G[M, K]}$$

6: Add $P(S)$ to bin corresponding to R in $\mathcal{P}$
7: **end for**
8: **return** $\mathcal{P}$

fig. 4.4. Algorithm to calculate response time probability distribution using normalization constant.

AI PERFORMANCE VS KPI

We have already identified the central AI server as the system's bottleneck. In this chapter, the goal is to further analyze possible scenarios of onboard AI behavior, model these scenarios, incorporate them into the broader queueing network (QN) framework, and ultimately evaluate how they impact the overall KPI.

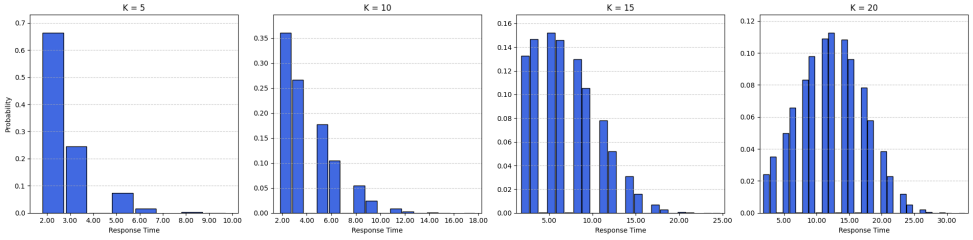

fig. 4.5. Calculated response time probability distribution.

ONBOARD AI

To reiterate, the UAV is equipped with a first layer of AI software capable of detecting unusual objects within its visual field. We assume this software operates both accurately and quickly. The next task is to classify each detected object into one of two categories: *threat* or *non-threat*. For example, a UAV may observe a bird during flight; this object should be correctly identified as non-threatening. If it is mistakenly classified as a drone, this results in a **false positive**, leading to wasted time tracking the bird instead of continuing patrol and identifying real threats.

The possible outcomes when a UAV detects an object are:

- **Real threat**
  - Correctly tracks the threat → KPI gained (**true positive**)

- Fails to track the threat → KPI lost (**false negative**)

- **Non-threat**
  - Correctly ignores the object → no KPI impact (**true negative**)
  - Incorrectly tracks the object → KPI lost (**false positive**)

Using this classification framework, we can now derive two formulas to quantify the rate of false negatives and false positives.

$$P(FalsePositive) = (1 - A_0) * (1 - r)$$
$$P(FalseNegative) = (1 - A_0) * r \tag{4.7}$$

where A_0 is accuracy of onboard AI and r is fraction of real threats among all objects.

FALSE CLASSIFICATION IMPACT

Each time the UAV detects an object, a request is sent to the central AI. If the onboard classification is correct—regardless of whether the object is a threat or a non-threat—the central AI's response will match the onboard AI, and no KPI will be lost. However, we are interested in the opposite cases, where the onboard AI misclassifies the object.

In the first case, the UAV begins tracking a non-threat. Once it receives the correct classification from the central AI, it will stop monitoring the object and resume searching for actual threats. Here, some time is lost while the UAV was incorrectly monitoring a non-threat. In the second case, the UAV fails to detect a real threat. Upon receiving the corrected classification from the central AI, it will redirect to the threat. Again, KPI is lost due to the delayed reaction.

Require: Response time PMF PMF, onboard accuracy A_0, threat ratio r, threat rate λ, simulation time T_{sim}
Ensure: Array of lost time fractions L
Set number of samples $S \leftarrow 5000$
$N \leftarrow \lfloor \lambda \cdot T_{\text{sim}} \rfloor$
$n_{\text{real}} \leftarrow \lfloor N \cdot r \rfloor$
$n_{\text{nonthreat}} \leftarrow \lfloor N \cdot (1 - r) \rfloor$
$n_{\text{fp}} \leftarrow \lfloor (1 - A_0) \cdot n_{\text{nonthreat}} \rfloor$
$n_{\text{fn}} \leftarrow \lfloor (1 - A_0) \cdot n_{\text{real}} \rfloor$
Extract (R, P) from PMF, where R is array of response times and P their probabilities
Normalize P: $P \leftarrow P / \sum P$
Sample $n_{\text{fp}} \times S$ values from R using P: FP_samples
$\text{wasted}_{\text{fp}} \leftarrow \sum$ FP_samples along each column
Sample $n_{\text{fn}} \times S$ values from R using P: FN_samples
$\text{wasted}_{\text{fn}} \leftarrow \sum$ FN_samples along each column
$\text{wasted}_{\text{total}} \leftarrow \text{wasted}_{\text{fp}} + \text{wasted}_{\text{fn}}$
$L \leftarrow \text{clip}(1 - \text{wasted}_{\text{total}} / T_{\text{sim}}, 0, 1)$
return L

fig. 4.6. Algorithm calculates percent of wasted time out of total simulation time. Wasted time is the time UAV spent tracking wrong object that is incorrectly classified.

Our goal is to estimate the fraction of the total simulation time lost due to incorrect onboard AI classifications. By progressively improving either the onboard AI's accuracy or the central

AI's performance, we can generate plots showing the fraction of lost time as a function of each. These plots will help determine whether investment should be prioritized in improving onboard AI accuracy or in reducing the response time of the central AI.

Using the probability distribution data from the previous section and formulas (5.10) and (5.11), the final algorithm is presented below. By setting the total simulation time (e.g., 40 minutes), we can calculate the expected number of false positives and false negatives. For each such case, we sample a response time from the probability distribution and sum the durations. Finally, we calculate the fraction of total simulation time lost due to these delays (fig. 4.6). This value reflects the performance degradation caused by onboard AI errors.

First, we illustrate how the expected lost time decreases when the onboard AI accuracy is fixed at base 60%, while the central AI response time improves from 2 seconds to 0.5 seconds (left figure). One key observation is that the difference between the 95th percentile (p95) and the median (p50) is less than 2%. As the central AI performance improves, this difference becomes negligible—unlike in the case of onboard AI improvement, where the gap remains more pronounced.

The onboard AI improvement curve follows a linear trend, which aligns with the structure of the false positive/false negative probability formulas—specifically, the number of errors decreases linearly with increasing AI accuracy. In contrast, central AI improvement yields a curve that grows faster than linear, indicating a more significant gain in performance for each unit of response time reduction (fig. 4.7).

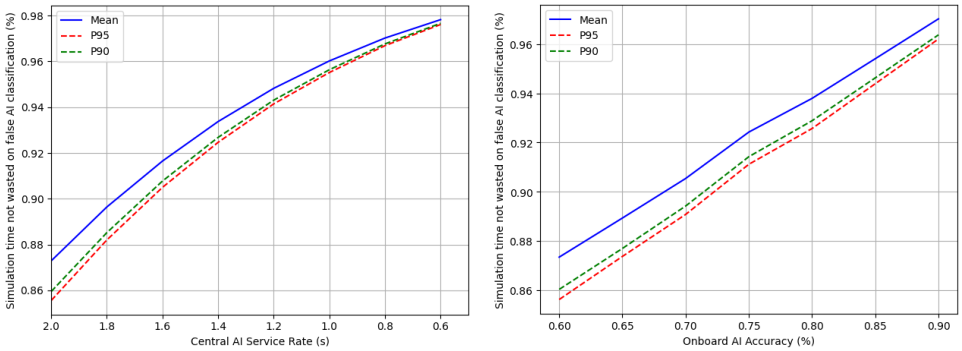

fig. 4.7. Plots show how reduction in wasted time depends on improvement of component performance. On the left – keeping base onboard AI accuracy at 60% and increasing central AI service rate. On the right – base central service rate at 2s and increasing onboard AI accuracy.

The plots below compare both strategies (fig. 4.8). As the object detection rate increases, the gap between the two lines becomes more pronounced. For example, when the base utilization of the central AI is 0.95, a 60% improvement in the central AI's processing rate results in 7% less wasted time compared to the same level of improvement in onboard AI accuracy.

Finally, it is important to note that the parameter r (the proportion of real threats) does not affect the plots. This is because it does not alter the total number of false positives and false negatives — only their distribution, so the overall time loss remains unchanged.

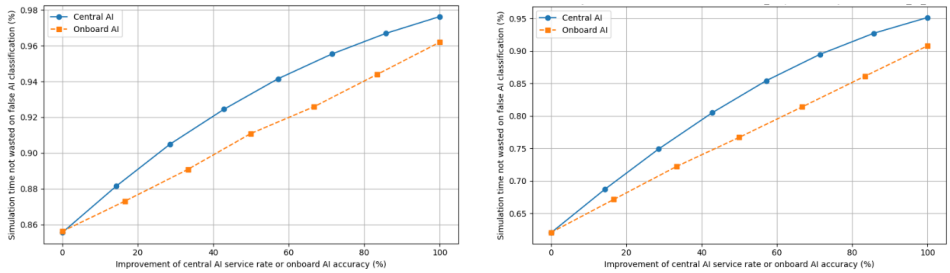

fig. 4.8. Combination of two plots from figure 4.7. Object rate is 3/min and utilization 78% (left), and 5/min, and 95% (right)

4.4. Conclusion

This chapter introduced a performance modeling framework for cooperative UAV threat detection missions using a **closed Gordon–Newell queuing network (QN)**. The model captures the dynamic behavior of a hybrid AI architecture decision pipeline, where both onboard and central AI participate in object classification, and in rare cases escalate to a human operator.

For the analysis of the QN model, baseline processing rate parameters were determined for each of the QN components. Depending on the size of the object's perimeter, different components were required, as the number of UAVs increased proportionally. The reference perimeter and UAV count used were the same as in Section 3 — a length of 1.2 km and 10 UAVs.

The strategy for estimating service rates was based on average values reported in existing literature, considering the suburban location of the patrolling area. For the central AI service rate estimation, existing literature was analyzed to identify suitable hardware and software configurations achievable within a budget of $10,000.

The rate of objects identified by local AI significantly impacts system load. Consistent with the scenario analyzed in Section 3 — a relatively harsh stress test — detection rates between 1/min and 10/min were considered, while the average saturation of real threats was maintained at a constant of 0.1.

The system's clear bottleneck is the central AI service, with a base service rate of 1.5 seconds. At 20 UAVs, the system's throughput reaches its maximum at 0.66 jobs per second. With 10 UAVs, the throughput is 0.42 jobs per second, the central AI utilization is 63%, the response time is 3.1 seconds, and the average queue length at the central AI is 1.35.

The ability to construct the system's response time distribution using calculations based on the normalization constant provides a more practical perspective. In practice, the average response time of a server does not provide a complete picture. In the reference setup, while the average response time is 3.1 seconds, the 95th percentile (p95) response time is 8 seconds. This

indicates that to effectively handle the reference load, further investment in central AI server performance is necessary.

To model the behavior of onboard AI classification, a linear model was developed to estimate the number of false detections. This model was then used to calculate the average expected time wasted by a UAV while waiting for a response from the service to correct its behavior. It was found that improving onboard AI accuracy leads to a linear reduction in wasted time. In contrast, improving the central AI service rate results in an even greater reduction, and this improvement follows a logarithmic pattern, even when central AI utilization is well below 100%. The higher the server utilization, the greater the impact of performance improvements on reducing wasted time.

5. SMART VEHICLE EMERGENCY CONSENSUS

This chapter explores the topic of consensus in vehicular networks. It begins with the Byzantine fault model, examining its requirements and surveying existing implementation approaches. The discussion then shifts to the more practical crash-stop consensus model, highlighting common use cases and performance challenges. Finally, a proposed method for modeling the performance of flooding-based consensus across various network topologies is presented, using NS-3 simulations supported by analytical approximations.

5.1. Introduction

6.1.1 Emergency situations

Two emergency road scenarios are discussed in this section: collision avoidance at an intersection and collision avoidance within a vehicle platoon. In figure 5.1, we see an intersection where a group of vehicles (Group B) travel normally along the main road. Another vehicle, A, is approaching at high speed and is unable to stop in time. Vehicles C and D are observing the behavior of the incoming vehicle A. To prevent a potential incident, we consider three theoretical approaches that could be used by intelligent vehicles:

- Each vehicle independently decides how to respond to the situation.
- Each vehicle makes a decision, followed by a voting process; the strategy with the most votes is selected and applied by the group.
- A consensus algorithm is executed, allowing all vehicles in the group to agree on a single coordinated strategy.

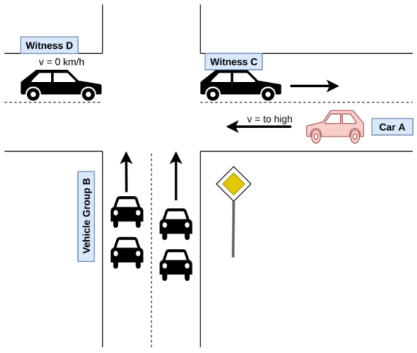

fig. 5.1. Example use cases of emergency crash avoidance consensus in V2V.

Let us now examine each option in detail. We assume that each vehicle is equipped with its own sensors to observe the environment. In the first approach, where each vehicle makes an independent decision, some vehicles in Group B may not detect the approaching vehicle A due to limited visibility or sensor range. As a result, certain vehicles might initiate an emergency stop, while others may continue driving, unaware of the threat. This uncoordinated behavior

could itself lead to a different, potentially more dangerous, type of accident. Therefore, this approach is not promising.

Now consider the second option: what if an authenticated central server collects the individual decisions of all vehicles? In this setup, any vehicle can retrieve the decisions, verify their authenticity via digital signatures, and compute a collective decision by applying a majority function to the collected inputs [46]. While this approach is theoretically viable, it depends on the presence of a trusted central server, most likely cloud-based—that acts as the source of truth. However, this introduces a single point of failure and demands highly capable infrastructure to process decisions for an entire city in real time.

What we truly need is an algorithm that ensures all vehicles reach the same decision simultaneously, despite each starting with different local observations. Moreover, this must happen without reliance on any privileged entity. What we have just described is the formal definition of *consensus*.

The second scenario is illustrated in the figure below (fig. 5.2). Here, a platoon of vehicles is driving on a highway when a speeding vehicle is suddenly detected ahead. While there may be several strategies to avoid a collision, the underlying nature of the problem remains the same as in the first scenario: vehicles must make a coordinated decision based on partial observations. The key difference lies in the topology of the vehicles—how they are positioned and connected—which becomes the main focus of interest in this research.

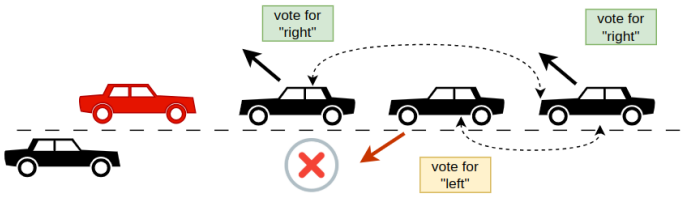

fig. 5.2. Emergency crash avoidance consensus during platoon.

5.2. Byzantine errors and Vehicular networks

In this chapter, we analyze whether Byzantine crash failures are relevant in vehicular scenarios. The definition of the Byzantine fault model was introduced in section 1.1 (fig. 5.3). This analysis aims to assess the likelihood and impact of such failures in the context of V2V communication and decision-making.

CAN IT BE OBSERVED IN REAL WORLD

There is a common misconception that Byzantine failures are purely theoretical and not worth addressing in real-world systems [47]. In practice, centralized architectures dominate the software industry due to their relative simplicity in implementation and maintenance. This observation is relevant because many decentralized or P2P systems fundamentally rely on Byzantine fault tolerance to function correctly [9]. In such systems, consensus among nodes is critical to ensure the reliability and correctness of the service. In contrast, centralized systems

typically follow a client-server model, where decision-making is handled solely by the server, and consensus mechanisms are not required. However, in recent years, there has been a growing emergence of decentralized architectures, including IoT networks and VANETs [48].Despite this trend, Byzantine failures are only partially addressed in existing research. Since V2V communication models inherently operate as distributed systems and rely on coordinated behavior, the study of Byzantine fault tolerance in this context is not only relevant but necessary.

The limited attention given to Byzantine fault tolerance in V2V systems remains speculative. One possible reason is the widespread belief that encryption, digital signatures, and public key infrastructures (PKI) alone are sufficient to secure a system [49]. For example, Cooperative Intelligent Transport Systems (C-ITS) standards focus extensively on PKI configurations but do not address fault tolerance at all [15]. Even the original Byzantine Generals paper [25] suggests that, with encryption and digital signatures, agreement can be reached regardless of the number of faulty nodes. However, later research has clarified that this claim depends heavily on the underlying network model—specifically, whether it is synchronous or asynchronous [50]. In asynchronous or partially asynchronous systems, there remain strict limitations on the number of faulty processes that can be tolerated. Moreover, even in synchronous networks equipped with digital signatures, consensus algorithms must still be executed to achieve agreement. Security mechanisms such as encryption and signatures address authenticity and integrity but do not replace the need for fault-tolerant coordination protocols.

In contrast, industries such as aviation not only acknowledge the importance of Byzantine fault tolerance but also actively research and integrate BFT solutions into their systems [51].As noted in [47], one reason BFT is often overlooked in mainstream system design is the assumption that "if I haven't seen it, it doesn't exist." The authors argue that this belief stems from limited exposure to faulty manifestations in real systems. They illustrate the point with a striking example: "However, digital designers do not have (and cannot have) enough 'hands-on' experience with hardware to make such a statement. If a designer spent 50 hours per week, 52 weeks per year, for 35 years staring at one system, that would be less than 105 hours... far short of typical avionics requirements" [47]. This highlights the gap between real-world exposure and the level of reliability demanded in safety-critical systems—further justifying the need for BFT, especially in domains like V2V communication, where safety is also paramount.

VEHICULAR AD-HOC NETWORKS

In this chapter, we discuss the fundamental characteristics of VANETs. The concept of inter-vehicle communication originated in the early 2000s, and VANETs have since been developed based on the principles of MANETs. VANETs represent a subfield of ITS, encompassing not only mobile ad hoc networking among vehicles but also broader system components such as vehicle-to-infrastructure communication RSU, RSU-to-traffic center communication, and more.

VANETs support a wide range of applications, from simple one-hop message dissemination to complex multi-hop communication over longer distances. While many concerns in MANETs also apply to VANETs, the specific mobility patterns and communication requirements of

vehicles introduce important distinctions [52]. For example, vehicle movement typically follows structured traffic patterns rather than random mobility models. Communication is often implemented using short-range wireless protocols such as Dedicated Short-Range Communications (DSRC). The IEEE 802.11p standard, an amendment to 802.11a, was introduced in 1999 to support V2X communication, featuring modified PHY parameters and adjusted MAC contention window sizes. Operating in the 5.9 GHz DSRC band, 802.11p uses half the bandwidth of 802.11a while maintaining compatibility with its modulation schemes [13]. In North America, DSRC transceivers have been mandatory for newly manufactured vehicles since 2016 [53].

The primary goal of V2V communication is to support safety-critical applications such as electronic brake warnings, platooning, traffic condition alerts, emergency response coordination, and on-road services. In such safety-critical systems, the risk of Byzantine failures cannot be ignored. These types of faults may occur at rates far exceeding the thresholds common in aerospace systems, typically 10^{-6}, 10^{-9}, or even 10^{-12} failures per hour [47]. To illustrate, consider that there are approximately 292 million registered vehicles in Europe [54]. With an average annual mileage of 12,000 km per vehicle [15], and assuming an average speed of 50 km/h and a Byzantine hardware fault every one million hours, the estimated rate of hardware-induced Byzantine failures would be approximately 1,300 byzantine errors per week across Europe.

EXISTING WORK ON BYZANTINE ERRORS IN V2V

In this chapter, we provide an overview of existing research on BFT in VANETs. The authors of [55] argue that the reliability of VANETs can be enhanced by applying BFT algorithms to decision-making processes within ad hoc networks. Additionally, they suggest that many challenges in VANETs can be mitigated through the use of clustered topologies, which are recommended as a strategy for addressing congestion. The study also assumes the presence of a PKI, as indicated by the statement: “The PE can always identify the sender of the information,” implying that message authentication is a given. Overall [55] ,surveys a substantial body of related research and proposes an extended version of Lamport’s algorithm, which includes “leave” and “join” functionality to support dynamic network membership. However, the primary limitation of this work is that it does not address the message complexity of the BFT algorithm. In particular, if the underlying algorithm resembles the original Lamport protocol, the number of messages grows exponentially with the number of nodes—making it potentially unsuitable for highly dynamic or large-scale VANET environments.

In [56], the authors propose a protocol for VANETs that prioritizes user privacy and introduces incentive-based event message dissemination. They highlight two core challenges: “First, it is difficult to forward reliable announcements without revealing users’ identities. Second, users usually lack the motivation to forward announcements.” This observation underscores that users are generally unwilling to participate if their privacy is compromised and there is no tangible benefit. The proposed solution is a privacy-preserving announcement protocol that employs threshold signatures to protect user identities.

Specifically, the authors introduce the concept of Anonymous Group Participants (AGPs), whose event messages are the only ones propagated across multiple stations and received by various vehicles. The privacy of AGP generators is safeguarded using a threshold ring signature scheme. Although the protocol incorporates a blockchain-based incentive mechanism, the justification for using blockchain—namely, coin generation as motivation—remains unclear, especially since the blockchain is not public and coin generation is a privileged operation. Furthermore, while the paper addresses privacy and incentives, it does not consider the issue of reaching agreement among vehicles in response to road events. The overall solution is relatively complex, involving advanced cryptographic constructs such as threshold signatures and consensus algorithms, without explicitly addressing coordination in emergency scenarios.

In [57], the authors present a novel proof-of-eligibility mechanism for achieving consensus in VANETs. The core idea is to restrict participation in consensus to a subset of nodes directly involved in a specific event. This approach is intended to prevent nodes within short communication ranges from compromising the consensus process by increasing the likelihood of exceeding the one-third threshold of malicious participants, as outlined in the Byzantine fault model [25].

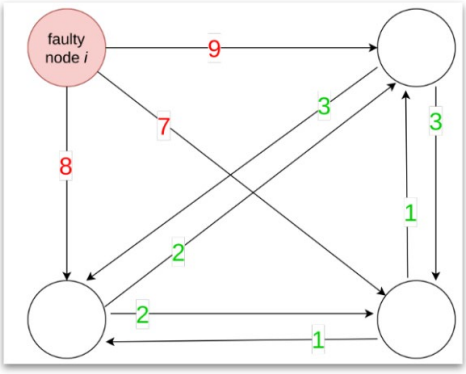

fig. 5.3. Example of the byzantine error during consensus. Malicious node broadcast inconsistent values.

When an event occurs, the first reporting node generates a message that includes a challenge - a set of questions related to the event—and a hash of the correct answers. Any node that wishes to participate in the consensus must provide the correct answers to this challenge. If the resulting hash matches, the node is allowed to derive a secret key. Only nodes possessing this key can proceed to the consensus phase. The authors argue that traditional consensus algorithms are unsuitable for highly dynamic networks like VANETs, as they typically require a fixed set of participants. Instead, they propose an alternative consensus mechanism in which more than two-thirds of the participating nodes must agree on both the membership list and the event value.

However, the rationale for using the “more than two-thirds” threshold is not clearly justified. While this threshold is standard in Lamport’s synchronous, oral-message consensus model, the system in question assumes signed messages and operates under an asynchronous

model. Moreover, the consensus algorithm itself is not fully specified. Although the authors claim that their approach does not require a trusted facility, it still relies on the existence of a PKI, implying some level of centralized trust.

In [58], the authors address the problem of false data injection in vehicular network event reports. While numerous mechanisms have been proposed to enhance the security of VANETs, many remain susceptible to the injection of false information, which can be used to manipulate the behavior of other drivers. To mitigate this, the authors propose an algorithm based on the concept of *Proof of Relevance* (PoR), which aims to demonstrate that the reporting node is genuinely related to the event being reported.

The PoR mechanism is achieved through what the authors refer to as *authentic consensus*. In this process, the event reporter collects signed opinions from nearby witnesses to form a final certificate confirming the event's validity. This authenticated report is then disseminated to other road users. A key challenge in this approach is the efficient collection of signatures, as naïvely broadcasting signed messages can result in excessive duplication and significant network overhead. As noted by the authors, "simply disseminating the signed messages results in many duplicate transmissions that waste a lot of time and network resources."

Message verification, however, is straightforward: if a vehicle receives at least T valid signatures, it accepts the event as trustworthy and acts accordingly. The parameter T can be chosen dynamically based on the type of event, or the selection process can be delegated to a trusted authority.

5.3. Fail stop consensus

In this chapter, we shift focus from the Byzantine fault model to the more practical crash-stop failure model. The research questions will first be defined, followed by a discussion of existing approaches, the method proposed in this thesis, the results, and the final conclusions.

In contrast to the Byzantine failure model, where processes may behave arbitrarily and even maliciously, this research adopts the fail-stop model. In this model, algorithms are designed under the assumption that processes may fail by crashing, but such failures can be reliably detected by all other processes [1]. The same use case of emergency consensus, introduced previously, will be analyzed under this more realistic failure model.

THE PROBLEM

This research focuses on a few important questions related to using fail-stop consensus in V2V communication during emergencies. First, can a fail-stop consensus algorithm work well in urgent V2V situations where vehicles need to quickly agree on what to do? Second, how does the number of vehicles in the network affect how fast and well the algorithm works? Third, how does the network topology affect the algorithm's performance? And finally, can a flooding-based approach, where messages are quickly shared with all nearby vehicles, be a good way to reach agreement in these emergency cases?

FLOODING CONSENSUS

This chapter focuses on the analysis of a specific algorithm from [1]—the flooding consensus algorithm:

```
upon event ⟨ c, Init ⟩ do
    correct := Π;
    round := 1;
    decision := ⊥;
    receivedfrom := [∅]^N;
    proposals := [∅]^N;
    receivedfrom[0] := Π;

upon event ⟨ P, Crash | p ⟩ do
    correct := correct \ {p};

upon event ⟨ c, Propose | v ⟩ do
    proposals[1] := proposals[1] ∪ {v};
    trigger ⟨ beb, Broadcast | [PROPOSAL, 1, proposals[1]] ⟩;

upon event ⟨ beb, Deliver | p, [PROPOSAL, r, ps] ⟩ do
    receivedfrom[r] := receivedfrom[r] ∪ {p};
    proposals[r] := proposals[r] ∪ ps;

upon correct ⊆ receivedfrom[round] ∧ decision = ⊥ do
    if receivedfrom[round] = receivedfrom[round − 1] then
        decision := min(proposals[round]);
        trigger ⟨ beb, Broadcast | [DECIDED, decision] ⟩;
        trigger ⟨ c, Decide | decision ⟩;
    else
        round := round + 1;
        trigger ⟨ beb, Broadcast | [PROPOSAL, round, proposals[round − 1]] ⟩;

upon event ⟨ beb, Deliver | p, [DECIDED, v] ⟩ such that p ∈ correct ∧ decision = ⊥ do
    decision := v;
    trigger ⟨ beb, Broadcast | [DECIDED, decision] ⟩;
    trigger ⟨ c, Decide | decision ⟩;
```

fig. 5.4. Fail-stop flooding consensus [1].

The algorithm on figure 5.4, referred to as the flooding consensus algorithm, provides the following guarantees:

- **C1: Termination** — Every correct process eventually decides on some value.
- **C2: Validity** — If a process decides on a value v, then v must have been proposed by some process.
- **C3: Integrity** — No process decides more than once.
- **C4: Agreement** — No two correct processes decide on different values.

The algorithm operates under the assumption of a perfect failure detector. In the absence of failures, it completes in a single communication step, with all processes deciding at the end of the first round. Each crash failure can delay termination by at most one additional round. Therefore, in the worst-case scenario—where $N-1$ processes crash sequentially—the algorithm requires up to N rounds to complete.

In each round, the algorithm incurs $O(N^2)$ message complexity. Additionally, after a process reaches a decision, it broadcasts $O(N^2)$ DECIDED messages. For each extra round triggered by a crash, another $O(N^2)$ messages are exchanged. Consequently, in the worst case, the total number of messages can grow to $O(N^3)$.

METHODOLOGY

The V2V context is highly dynamic, with many factors influencing system reliability. Flooding consensus algorithms assume an upper bound on message delivery time and reliable, time-bounded crash detection. However, false failure suspicions—where a correct process is mistakenly believed to have crashed—can violate the agreement property. In such cases, two correct processes may decide on different values, breaking the consistency of the protocol. Conversely, if a crashed process is not suspected (i.e., the failure detector violates the strong completeness property), the termination property of consensus may be compromised, as some processes could wait indefinitely.

In the V2V environment, two primary factors affect reliability: wireless network interference and the absence of full connectivity among consensus participants. Because not all nodes in the group have direct communication links, multihop communication protocols must be employed. These protocols introduce additional complexity and potential delays, further challenging the assumptions made by traditional flooding consensus algorithms.

In this model, we simulate consensus among V2V nodes by representing the network as a connected graph, where each node broadcasts messages that are flooded through the mesh. The flooding mechanism will be described later. Only links with delays below a predefined real-time threshold are included; links exceeding this threshold are treated as non-existent, modeling the constraints of real-time communication (fig. 5.5). This abstraction allows us to assume a perfect failure detector within the connected graph, as all the paths included are fast and reliable. Nodes unreachable within the threshold are considered failed for the purpose of consensus. This approach enables estimation of consensus delay in highly dynamic V2V environments while maintaining a tractable simulation model. The remaining task is to construct a simulation to verify whether a reliable and low-latency mesh network can be established over a simple graph of wireless V2V nodes.

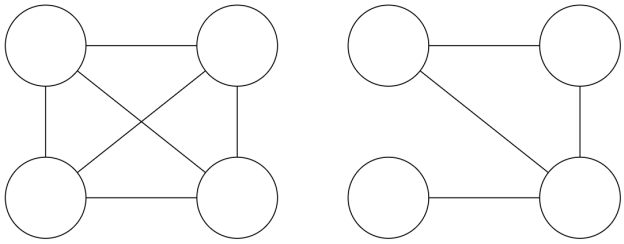

fig. 5.5. Complete graph (left) and connected graph (right). Edge means there is bounded reliable connection and missing edge means either no connection at all or delayed delivery.

HWMP PROTOCOL

The Hybrid Wireless Mesh Protocol (HWMP) is a routing method used in certain types of wireless networks called mesh networks, where devices connect to each other directly without relying on a central router [59]. HWMP decides how data should travel across the network by combining two techniques. The first, called proactive routing, means that some routes are

always kept ready in advance, like keeping a map of known paths, so data can move quickly without delay. The second, called reactive routing, means that other routes are created only when needed, which saves effort when no data is being sent. HWMP uses both methods together: it keeps main routes prepared ahead of time and finds other routes on demand, helping the network run smoothly and efficiently.

HWMP was chosen for this study as a baseline routing protocol due to its standardization in IEEE 802.11s and built-in support in NS-3, ensuring reproducibility and reducing implementation complexity. While HWMP is not specifically designed for highly dynamic V2V communication, the focus of this research is on consensus performance in emergency scenarios such as crash avoidance, where vehicles often operate in platoons or localized groups moving at similar speeds. In such contexts, the network topology is relatively stable over short durations, making HWMP's routing behavior sufficiently reliable. Using HWMP also allows for evaluating consensus mechanisms under realistic but suboptimal network conditions, serving as a conservative benchmark for future work involving V2V-optimized protocols.

Enhanced HWMP (E-HWMP) refers to various proposed extensions to the original Hybrid Wireless Mesh Protocol aimed at improving its performance in highly dynamic environments such as VANETs. These enhancements typically focus on increasing route stability, reducing control overhead, and improving responsiveness by incorporating mobility-aware metrics, geographic information (e.g., GPS coordinates), or adaptive route maintenance strategies. E-HWMP variants are designed to address the limitations of standard HWMP in scenarios with frequent topology changes, such as high-speed vehicle movement, where traditional HWMP may struggle to maintain valid routes. While not part of the IEEE 802.11s standard, E-HWMP approaches demonstrate how HWMP can be adapted for more demanding V2V use cases.

SIMULATION

The primary objective of the simulation is to measure how long it takes for all nodes in a wireless mesh network to reliably broadcast their unique messages, such that each node eventually holds a complete vector of all other node IDs. The simulation accepts input parameters for topology configuration, number of nodes, packet size, and random seed. During execution, each node generates a unique identifier and broadcasts it using UDP. To meet the specified packet size, random padding bytes are appended to the ID.

To evaluate broadcast propagation performance in wireless mesh networks, a custom simulation was developed using the NS-3 network simulator. The simulation models a grid-based mesh topology and implements a custom flooding algorithm to determine the time required for full message dissemination.

The network is constructed as a two-dimensional grid of stationary mesh nodes using IEEE 802.11s. Node layout is controlled through command-line parameters for grid dimensions and inter-node spacing. Nodes are placed uniformly and assigned static positions via the “ConstantPositionMobilityModel”.

At runtime, each node installs a UDP socket to receive broadcast messages. Since UDP does not guarantee delivery, redundancy is introduced through randomized retransmissions. Each node schedules a configurable number of broadcasts, each separated by a random delay

between 1 ms and a user-defined maximum. These delays help reduce interference. The optimal retry count and delay depend on the specific topology. For example, in a 3×3 grid (diameter 2, 9 nodes), using 3 retries and a 30 ms delay achieves a 92% success rate, with an average all-to-all broadcast time of 30 ms. In contrast, a 4×4 grid (diameter 3, 16 nodes) with the same settings results in 0% success. Increasing the delay to 120 ms yields a 100% success rate and an average delay of 90 ms.

Each broadcast payload includes the sender's node ID and random padding, with total size determined by packet size parameter. Upon receiving a packet, nodes extract the sender ID and store it in a local log, ignoring duplicates and self-originated messages. This enables tracking which messages were successfully received across the network.

To evaluate flooding effectiveness, the simulation records the timestamp of the first broadcast and the latest successful reception across all nodes. The flooding time is defined as the time difference between these two events. A run is considered successful if all nodes receive messages from all others (i.e., each node receives $N-1$ unique IDs). Otherwise, it is counted as a failure.

The simulation supports repeated runs using randomized seeds and reports the number of successful runs along with the average flooding time. This allows statistical analysis of performance across different topologies and configurations.

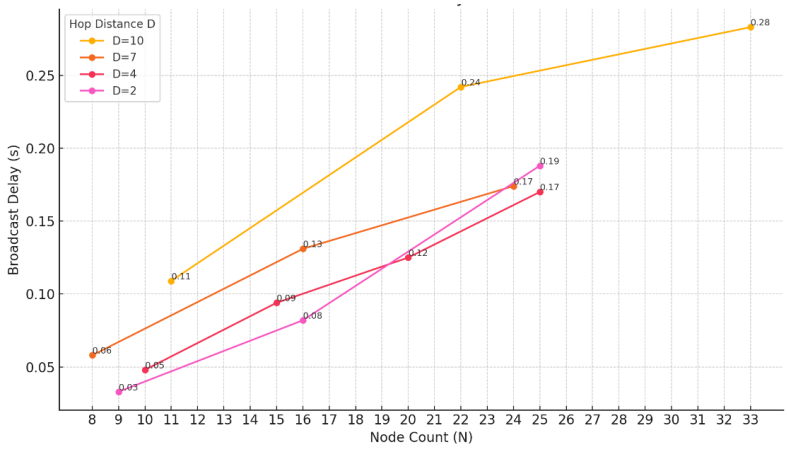

fig. 5.6. Average all-to-all broadcast delay aggregated from the simulation data.

This framework was used to investigate broadcast efficiency in small mesh networks under varying topological and operational conditions. Key parameters such as node count, inter-node spacing, jitter, retry count, and packet size were systematically adjusted to assess their impact on flooding time. The simulation provides a repeatable, controlled environment for analyzing mesh-wide information propagation.

EXPERIMENTS

The objective of the experiment is to determine the average all-to-all broadcast delay in wireless mesh networks. Simulations will be conducted for connectivity graph diameters of 2, 4, 7, and 10. The number of nodes will range from 8 to 33. For all experiments, the packet size will be fixed at 100 bytes; the impact of other packet sizes is left for future research.

Given that the current simulation environment constructs the network using a grid-based layout, achieving specific combinations of node count and connectivity diameter requires adjusting the inter-node distance. By varying this distance, different effective diameters can be simulated even with the same node count. The following figure (fig. 5.7) presents example configurations that satisfy selected combinations of node count and graph diameter.

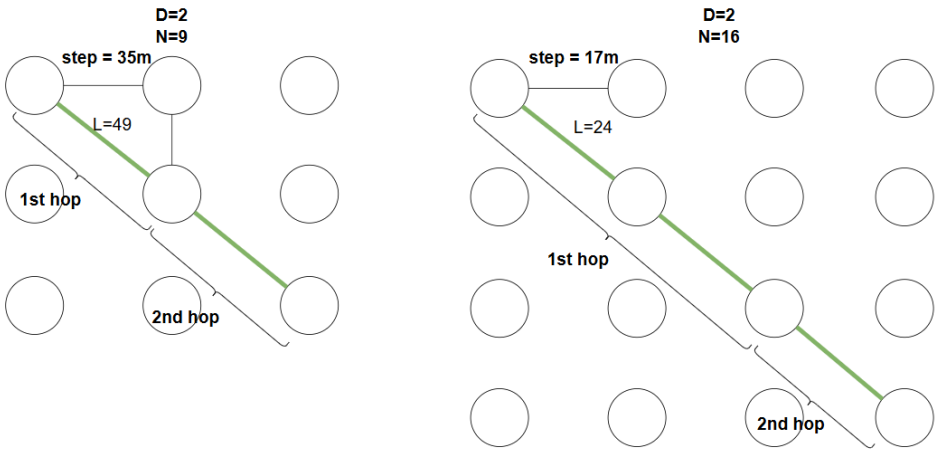

fig. 5.7. Examples of how to configure different combinations of N and D using variable steps. Communication range is constant (50m).

Table 5.1

All-to-all broadcast average completion time for different node count and diameters

D=10		D=7		D=4		D=2	
N	**T (ms)**	**N**	**T (ms)**	**N**	**T (ms)**	**N**	**T (ms)**
11	109	**8**	58	**10**	48	**9**	33
N/A	N/A	**16**	131	**15**	94	**16**	82
22	242	N/A	N/A	**20**	125	N/A	N/A
33	283	**24**	174	**25**	170	**25**	188

The simulation uses the UDP protocol without delivery acknowledgments. Given the strict timing requirements of emergency consensus—typically under 500 ms for full agreement—the focus of the simulation is to identify broadcast configurations that statistically guarantee delivery. In this model, when nodes execute consensus, each node broadcasts its message multiple times (e.g., three times), spaced by random delays. The expectation is that, with high

probability, the message will be successfully delivered to all other nodes. If a node fails to receive a message within this period, it is treated as having crashed.

Two key parameters influence the probability of successful message delivery: the retry count and the range of random delays between retries. For example, consider a configuration with a graph diameter of *D=4*, 15 nodes, 4 broadcast retries, and a retry delay range of up to 100 ms. The simulation is then executed 100 times using different random seeds. The goal is to identify configurations that yield a 100% success rate, that is, all 100 runs result in all nodes receiving all messages. If the success rate falls below 100%, either the retry count or the delay range is adjusted, and the process is repeated until a statistically reliable configuration is found.

The final Table 5.1 presents the results of statistically reliable all-to-all average broadcast timings for various connectivity graph diameters and node counts. The corresponding raw data is visualized in figure 5.6. Due to limitations in topology configuration flexibility, only three to four distinct node counts could be tested for each diameter. Nonetheless, the plots reveal a clear trend: for *D=2*, the broadcast delay increases with node count (positive derivative); for *D=4*, the growth appears approximately linear; and for *D=7* and *D=10*, the rate of increase slows down as node count grows, indicating a negative derivative. In the next section, we derive a mathematical approximation of this behavior.

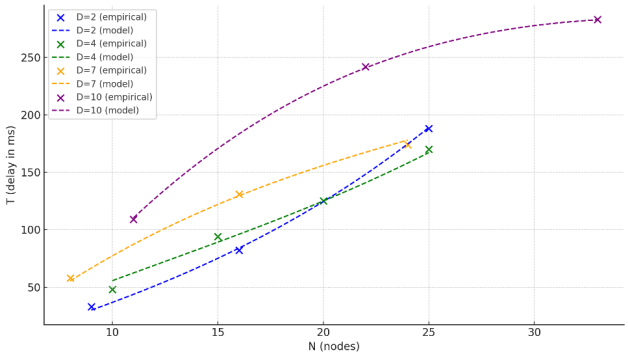

fig. 5.8. 3rd degree polynomial regression used to approximate simulation data.

5.4. Consensus execution time modeling

The next step is to approximate the simulation results and derive an analytical function that models the all-to-all broadcast delay as a function of the number of nodes *N* and the connectivity graph diameter *D*.

APPROXIMATION OF ALL-TO-ALL DELAY

To approximate the delay as a function of the number of nodes *N* and network diameter *D*, a third-degree **polynomial regression** model was used. Regression is a statistical method for finding the mathematical relationship between input variables and an output value — in this case, predicting delay *T* based on *N* and *D*. Polynomial regression extends linear regression by

including higher-order and interaction terms, which allows the model to capture nonlinear effects.

To train the model, all available data points were first combined into a dataset of *(N, D, T)* triples. Then, using the least-squares method, the model determined the optimal coefficients that minimize the error between the predicted and actual delay values. This resulted in a fitted polynomial function including terms like N^2, ND, D^3, etc., which reflects how delay changes with both variables. The final function was implemented as a Python function and visualized to confirm its close alignment with the empirical data (fig. 5.8).

The resulting analytical approximation of the all-to-all broadcast delay, based on the simulation data, is expressed by the following formula:

$$\begin{aligned} T(N, D) = {} & 7.783499 \cdot N + 16.127333 \cdot D - 0.090524 \cdot N^2 \\ & + 0.335272 \cdot ND - 2.444790 \cdot D^2 + 0.009286 \cdot N^3 \\ & - 0.072579 \cdot N^2 D + 0.189701 \cdot ND^2 + 0.010002 \cdot D^3 \\ & - 63.443260 \end{aligned} \tag{5.1}$$

Where N – number of nodes and D – diameter of the connectivity graph.

MODELING FULL CONSENSUS PERFORMANCE

With the analytical function modeling the delay for a single communication step (i.e., all-to-all broadcast involving $O(N^2)$ messages), we can now estimate the total consensus delay as a function of the system's crash rate (fig. 5.9).

Require: Number of nodes N, crash rate crash_rate, delay per round one_step_time
Ensure: Total consensus time total_time_ms
1: $f \leftarrow N \cdot \text{crash\_rate}$
2: $rounds \leftarrow 1 + f$
3: $\text{total\_time\_ms} \leftarrow (rounds \cdot \text{one\_step\_time}) + \text{one\_step\_time}$
4: **return** total_time_ms

fig. 5.9. Python function that returns total consensus time based on number of nodes and crash rate.

This enables the generation of delay estimates across a wide range of scenarios, from low-diameter topologies to high crash rate conditions (fig. 5.10).

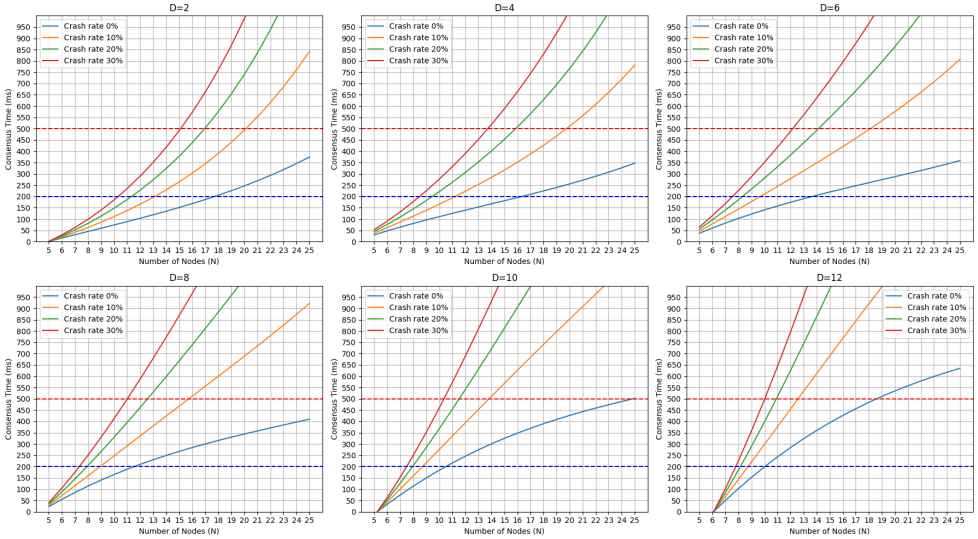

fig. 5.10. Approximated consensus time depending on nodes count and graph diameter.

5.5. Conclusion

During this research, the question arose as to whether Byzantine fault models should be considered in V2V communication when implementing consensus algorithms. To explore this, we reviewed experiences from other industries where Byzantine faults are treated as a serious concern. Existing BFT approaches in the context of V2X communication were also examined. The conclusion is that neither Byzantine nor other fault models are explicitly addressed in the official ITS documentation. This may be due to the low-level scope of such specifications or other practical considerations. In contrast, Byzantine fault models are mandatory in high-reliability domains such as avionics and spacecraft systems. The Byzantine fault-tolerant algorithms currently proposed for V2X systems in the literature tend to focus on privacy, blockchain-based architecture, and reputation mechanisms. However, no research has proposed an emergency consensus protocol for V2V that explicitly considers Byzantine faults.

Next, the question of whether V2V use cases can benefit from consensus algorithms was explored. As a reference, two emergency crash avoidance scenarios were considered: crossroad collision avoidance and crash avoidance within a vehicle platoon. The key idea is that consensus provides a guarantee of agreement among nodes, even in the presence of faults. This property can be particularly valuable in the scenarios mentioned, where consistent and timely decisions are critical for safety.

The main question was: what guarantees and assumptions must a consensus algorithm satisfy in the V2V context, and can it finalize quickly enough to meet the real-time constraints of emergency scenarios? The fail-stop flooding consensus algorithm from [1] was used as a reference. Classic flooding consensus assumes a perfect failure detector, which is unrealistic in V2V environments. This is because communication between nodes may experience variable

delays, leading to false crash detections. To address this, the V2V cluster was modeled as a simple, not fully connected graph, where a longer delay between nodes is treated as the absence of a link.

In this model, each node retains the interface of the classic algorithm, including the assumption of a perfect leader detector. However, at the lower network level, a multi-hop mesh topology is used to introduce redundancy and compensate for delayed links. In essence, longer communication delays are mitigated through alternative transmission paths and retry mechanisms. The goal of this approach was to quantify the cost of such a configuration in terms of lost messages and the delay of all-to-all broadcast operations.

A custom simulation and UDP-based flooding algorithm were developed using the NS-3 simulator to measure the all-to-all, multihop, statistically reliable broadcast delay in mesh networks with varying topologies and node counts. “Statistically reliable” means that, for each configuration, the average broadcast delay was computed only when the algorithm successfully completed 100 out of 100 runs—i.e., all nodes received all messages in every trial. The resulting raw simulation data was then used to perform polynomial regression, producing an accurate analytical function (Equation 6.1) that models the average all-to-all broadcast delay as a function of network size and structure.

Using the analytical function (5.1, a Python implementation was created to estimate end-to-end consensus delays, incorporating an additional crash rate parameter (). Plots were generated for all combinations of network diameter, node count, and crash rate. The goal was to identify the worst-case upper bounds for end-to-end consensus delays under varying conditions, particularly for higher crash rates, larger diameters, and greater node counts. These estimates are based on both the collected simulation data and its polynomial approximation. Boundary results show end-to-end consensus under 500 ms is achievable even with 30% crash rates (Table 5.2)

Table 5.2

Maximum node count for various diameters and limited maximum consensus completion time

D	N (Sub 500ms e2e delay)	N (Sub 200ms e2e delay)
2	15	10,2
4	13,5	8,5
6	12	7,5
8	11	7,3
10	10,3	7,3
12	10	7,3

The findings from this research prove that it is possible to implement statistically reliable all-to-all broadcast in V2V mesh networks using NS-3 simulation, leveraging HWMP, UDP, randomized retries, and connected graph topologies. The simulations showed that, for up to 33

nodes with a connectivity graph diameter of 10, reliable broadcast can be achieved in under 300ms. By approximating the all-to-all delay function, it becomes possible to extrapolate the expected delay for synchronous flooding-based consensus. Even in challenging conditions, for example, a platoon with a diameter of 10, nine nodes, and a crash rate of 30% — consensus can still be reached in under 350ms. The shape of the reliable broadcast delay curve depends on the diameter of the communication graph: with $D=2$, the delay grows positively with node count, while for $D\geq 7$, the growth begins to decrease. A more linear trend is observed around $D=4$. For example, with $N=9$ and a crash rate of 20%, the consensus delay is 120 ms for $D=2$, and increases to 250ms for $D=7$, representing a 108% increase.

These results offer indicative upper-bound delay values for a custom fail-stop flooding consensus algorithm assuming a perfect failure detector but operating over a network with potentially missing links. The estimates are based on NS-3 simulations using a custom UDP flooding algorithm built on top of HWMP. However, the current research does not consider additional messaging overhead (e.g., heartbeat messages) or dynamic node behavior, which are important factors in more realistic V2V scenarios. Future work may extend the model to incorporate these factors for a more complete understanding of consensus feasibility under real-world conditions. A complete NS-3 simulation that implements the full consensus algorithm—not just the broadcast layer—could also be beneficial for future research, enabling more detailed evaluation of timing, message complexity, and reliability under realistic V2V conditions.

CONCLUSION

The last decade has been marked by numerous technological and scientific breakthroughs. Among them, two areas stood out as the foundation for this work: the mass production of small aerial vehicles and the advancement of distributed systems. A major source of inspiration was the intersection of these fields through the application of smart contracts and robotics. The first published article was titled “Decentralized Byzantine Fault Tolerant Proof of Location” [60]**.** Blockchain technology is an integral part of distributed systems theory, though not always in its traditional form. It demonstrates that fault-tolerant algorithms can take on entirely new shapes, unlike those described in classical literature. For instance, network participants can be represented not as individual nodes but as computing power. This realization provided strong motivation to explore the application of this theory beyond virtual networks and finance, extending it to cyber-physical systems. One example is creating logistics chains that operate without third-party arbitration. However, this direction was challenging to pursue due to the lack of real-world demand for such futuristic technologies. Perhaps in the future, this field will once again attract followers.

Thus, the focus of this work shifted towards a more applied direction — namely, cooperative mobile networks. With extensive experience in developing software for large IT systems—where distributed systems theory is frequently applied — it became intriguing to explore whether this theory extends to networks with mobile intelligent nodes. Such nodes can be various devices, but smart vehicles and drones were of the greatest interest. The problem was that, even in relatively simple corporate IT solutions, distributed systems theory was often ignored, leading to costly failures and significant drops in product quality. Comparing the complexity of solutions for smart vehicles that can work together with that of a corporate system is challenging. However, the former is also a distributed system, but with software that is mobile and operates in an unreliable wireless network. This introduces new degrees of freedom into the system, which undeniably complicates the solution. The situation is even more complex with drones. Thanks to the department, I became familiar with real-world scenarios involving both drones and smart vehicles. One particularly interesting example is the use of drones for wind turbine quality inspection. This solution allows for detailed scanning of the turbine in a short time to detect even the smallest cracks. Everything happens autonomously, without human intervention. Such a solution can be further accelerated by adding a few more drones working together.

Another real scenario is video surveillance using drones. The challenge here is that many solutions and studies are proprietary, and the topic of drones is closely tied to the military industry, where research is also rarely published. In corporate systems that use distributed systems theory, fault tolerance is introduced to ensure resilience against various types of errors, which often occur in large and unexpected quantities. Errors can lead to financial losses, but in the scenarios described above, they can have even more severe consequences.

One of the most important algorithms in distributed systems theory is consensus. It can take many forms, ranging from fail-stop leader election to Byzantine total order broadcast and blockchain-based consensus. Consensus is also essential for the cooperation of drones and

smart vehicles. Based on all of the above, the objective of this work was formulated as follows: Research existing consensus algorithms, assess their applicability in advanced cooperative mobile network scenarios, and develop simulations to evaluate their performance under harsh conditions for potential development in real deployments.

The dissertation is built upon a series of studies, each of which helps answer specific questions posed to achieve the main objective. **In the first chapter**, the scenarios of cooperative mobile networks, which became the focus of the research, are described in detail:

- Consensus among smart vehicles to avoid collisions.
- Leader election among drones ensuring perimeter surveillance.

This chapter also briefly introduces the theoretical algorithms on which the rest of the work is based. Finally, it discusses existing research on these topics.

The second chapter is dedicated to the design and development of a simulation for studying fault-tolerant algorithms in various cooperative drone missions. Before the design process began, the main objectives were defined. The first objective was to ensure the reusability of the simulation code in real systems. This applies to almost all code layers: communication, distributed system primitives, message routing within drones, mission logic, and message processing. This focus is due to the expensive nature of drone equipment and the difficulty of testing all possible scenarios in real-world conditions. Thus, the simulation allows for testing code ranging from rendezvous and NAT traversal to leader election under failures and regrouping. The second objective was to focus the simulation on the realistic behavior of the distributed system rather than on detailed modeling of drone movement and the surrounding world. For example, 3D visualization can be sacrificed for a 2D simulation, but the nodes must communicate using a real network stack and a proxy server that simulates a poor network—introducing delays, out-of-order delivery, and packet loss.

Based on the main objectives, the option of using existing simulation platforms was discarded because none offered the use of a real communication stack, as most simulations focus on determinism. As a result, the platform developed within this dissertation can be considered a real-time simulation, meaning that if a scenario lasts five minutes, the simulation results will also take five minutes to generate. This has both advantages and disadvantages. In total, the development and debugging process took more than six months. Ultimately, all objectives set for the simulation were successfully achieved. As expected, the simulation revealed numerous bugs related to conflicts in parallel message processing and handling messages that arrived out of order.

The third chapter is devoted to the first scenario of applying cooperative mobile networks and fault-tolerant algorithms—specifically, the leader election problem. In essence, leader election is a special case of consensus because drones must agree on a single leader, and ideally, that leader should not be non-functional. Otherwise, the mission will be executed sub-optimally. Approaches to leader election can be divided into two major categories: those assuming synchronous communication and those assuming partially synchronous communication. Within both categories, there are numerous implementations with various additional assumptions, such as the requirement that the majority of nodes must be online. The main question of this part of the dissertation was: How can the impact of using an eventual consensus algorithm on mission

performance be measured, and which approach is optimal in the context of perimeter patrol by drones?

Perimeter patrol is a task where it is crucial to maintain consistent data about the ongoing mission; otherwise, two drones, for example, might end up patrolling the same area. Eventual leader election algorithms allow prioritizing the continuation of the mission, even under conditions of incomplete or inaccurate data. However, it is far from obvious how this will affect the mission. To address this, this work proposes a new method for "measuring" the impact of an eventual algorithm. The simulation introduces the concept of mission performance indicators — numerical metrics such as the number of threats captured on camera near the perimeter or the percentage of time at least one drone was monitoring an object.

Within this chapter, numerical results were obtained that reveal a great deal about the properties of the algorithms. The cumulative number of false positives triggered by the eventual algorithm during the simulation follows a logarithmic pattern. Only a small fraction of these false positives actually affects the KPIs. Despite the fact that the eventual algorithm loses 4.1% of the KPI compared to a theoretically ideal synchronous algorithm, it still outperforms a synchronous algorithm with a realistic configuration by an average of 9.7% in a 5-minute simulation with a 30% drone failure rate.

The fourth chapter is dedicated to the same perimeter patrol scenario, but the impact on KPI is considered from a different perspective. An integral part of such a system is AI-based services that help identify potentially dangerous objects around the perimeter and accurately classify them. For example, a drone's camera may capture birds or other animals, humans, or—at the extreme—enemy drones. It is assumed that the drone has an onboard AI service, but with limited accuracy due to the drone's hardware constraints. It is also assumed that there is a high-performance remote service with significantly higher accuracy and speed. The performance of this remote service is dictated by various parameters. The main question posed in this study was: Which has a greater impact on mission KPI — the performance of the central AI or the accuracy of the onboard AI? To address this question, a Gordon-Newell Closed QN model was proposed, allowing the entire network to be simulated—from drones to operator consoles. This model not only enabled the identification of the network's bottlenecks but also, using the approach proposed in the work, made it possible to calculate the theoretical probability distribution of system response time. Using this data, the model was extended to answer the main question. For this, the number of false positives from the local AI was represented as a simple linear model. Then, using the system response time distribution, it became possible to calculate the expected average time spent on false tasks by the drone. This time is assumed to have a direct impact on the KPI.

The calculation results show that even though the utilization of the central AI can reach around 70%, its linear performance improvement always provides more than a linear gain in terms of reduced lost time. In contrast, improving the accuracy of the onboard AI always results in a linear reduction in lost time. The higher the expected utilization of the central AI, the more advantageous it becomes to invest in its speed rather than in the accuracy of the onboard AI.

The fifth chapter is entirely dedicated to the second research scenario—consensus among smart vehicles to avoid collisions. Historically, the research began specifically with the

Byzantine class of faults. The first question posed was: How relevant is this class of faults in V2V networks? Interestingly, this class of faults has been known since the late 1970s, but at that time, their main cause was primarily related to unreliable computing hardware. Today, Byzantine faults are most relevant in open networks like blockchain, where it is logical to assume that without access control, powerful attacks such as Sybil Attacks and others are possible. The study showed that Byzantine faults are also seriously considered in avionics, space, and military technologies, where the cost of an error can be infinitely high. Existing literature on communication and consensus in V2V networks was reviewed. Notably, the ETSI ITS technical reports did not mention the consensus problem at all. In scientific papers, there were studies related to Byzantine faults, but their focus was more on privacy and blockchain rather than consensus for collision avoidance. Statistical analysis, taking into account data on the probability of Byzantine faults (statistics from the aviation industry) and the number of vehicles on the roads, shows that such faults can occur on European roads up to 1,000 times per week.

The second part of this study focused on a more practical class of faults—complete system failure. It is assumed that during an emergency situation, vehicles must quickly adopt a common strategy to avoid a collision. The basic algorithm works as follows: each node has its own data about the environment, and each node sends this data to every other node. Based on the combined data, a common and identical strategy can be derived. The key aspect here is the identical strategy because, similar to drone missions, if each node has a different view, the vehicles may behave sub-optimally or even dangerously. The main question of this study was: Can fail-stop flooding consensus be used effectively in this scenario? To answer this, an NS3 simulation was developed, along with an algorithm for broadcasting messages in an 802.11s mesh network using UDP. The simulation showed that the algorithm can provide reliable all-to-all broadcast in a network of 33 nodes with a diameter of 10 in 99% of cases, with an average delay of 280 ms. An approximation of the simulation data made it possible to represent the function of all-to-all broadcast time (one step of the consensus algorithm) depending on the number of nodes and the network diameter. Further calculations were performed to determine the average consensus time based on the number of nodes, diameter, and number of failures. It is important to note that a diameter of 10 steps does not represent the actual network topology but rather models a more realistic connection graph where any delay longer than X ms is considered an absence of an edge in the graph.

The calculation results showed that a group of nine vehicles, with a graph diameter of 10 and a 30% failure rate among vehicles, can still achieve consensus in an average of 350 ms. The shape of the reliable broadcast delay curve depends on the diameter of the communication graph: with D=2, the delay increases positively with the number of nodes, while for D≥7, the growth begins to slow down. A more linear trend is observed around D=4. For example, with N=9 and a crash rate of 20%, the consensus delay is 120 ms for D=2, but increases to 250 ms for D=7, representing a 108% increase.

The doctoral thesis summarizes the results of completed research and defines possible **future research directions**:

1. Testing the software developed for the perimeter patrol study on real drones using a cellular network and the NAT traversal system also developed as part of the research.
2. Expanding the simulation's capabilities, such as enabling non-real-time simulation, implementing more complex coordination systems, and providing a more realistic network simulation.

The following main conclusions were obtained during the development of the doctoral thesis:

1. Real-time simulation is invaluable for identifying bugs that would otherwise appear during real-world deployment, where fixing them can be significantly more costly. The primary drawback of real-time simulation is its execution time. A hybrid approach—using discrete simulation for overall performance testing and real-time simulation for advanced corner cases in later project phases—can help achieve maximum fault tolerance and resilience.
2. Simulation data indicates that the cumulative number of false crash detections caused by the eventual leader detector algorithm during simulation follows a logarithmic pattern. As the number of deployed UAVs increases, the time until the last false crash detection also increases significantly. With more than five UAVs and a larger detection timeout (DT), false crash detections can continue for up to 10 minutes. Nevertheless, due to the logarithmic nature of this behavior, most false crashes occur quickly, and only a small portion of total false negatives affect KPIs.
3. In cooperative UAV perimeter patrol mission simulations, quantitative KPI metrics are valuable for comparing leader election algorithm performance and for determining whether eventually consistent algorithms are suitable for cooperative UAV scenarios.
4. In the "Perfect Leader Detector" algorithm, extending the crash detection timeout from 15 seconds (ideal) to 60 seconds results in a 12.1% KPI reduction during a five-minute simulation with a 30% UAV crash rate. The "Eventually Perfect Leader Detector," configured with a 5-second base crash detection timeout and a 3-second delta, shows a 4.1% KPI loss compared to the "Perfect" algorithm with the ideal timeout. Notably, the "Eventually Perfect" algorithm outperforms the "Perfect" algorithm under realistic conditions (60s timeout), achieving a 9.7% KPI improvement at a 30% crash rate.
5. A UAV surveillance network employing AI-based threat detection can be modeled using a Closed Gordon-Newell Queueing Network. While improvements in onboard AI accuracy reduce time lost to false positive monitoring linearly, reducing the central AI's response time has a greater impact on minimizing lost monitoring time under increasing system load.
6. The Byzantine error model is widely considered in avionics and space engineering. However, this term is never mentioned in ETSI ITS technical reports. A literature review of ITS reveals that Byzantine fault-tolerant algorithms proposed for V2X systems primarily focus on privacy, blockchain-based architecture, and reputation mechanisms. However, no research has yet proposed an emergency consensus protocol for V2V that explicitly considers Byzantine faults.

7. A custom randomized UDP-based reliable all-to-all broadcast algorithm using 802.11s mesh, tested with NS3 simulation, shows an average delay of 280 ms for 33 nodes with a connectivity graph diameter of 10. For smaller diameters and node counts, average delays are below 100 ms.
8. Mathematical modeling shows that the fail-stop flooding consensus algorithm, based on the aforementioned broadcast algorithm, can theoretically achieve consensus in under 500 ms for D=8, N=11, with up to 30% node crashes, and under 200 ms for smaller diameters. The shape of the reliable broadcast delay curve depends on the communication graph's diameter: for D=2, delay increases with node count, while for D≥7, the growth rate decreases. A more linear trend is observed around D=4.

The goal defined in the dissertation — to research existing consensus algorithms, assess their applicability in advanced cooperative mobile network scenarios, and develop simulations to evaluate their performance under harsh conditions for potential real-world deployment — was successfully achieved.

REFERENCES

[1] C. Cachin, R. Guerraoui and L. Rodrigues, Introduction to Reliable and Secure Distributed Programming, Berlin: Springer, 2011.

[2] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, San Francisco: Morgan Kaufmann, 2008.

[3] N. A. Lynch, Distributed Algorithms, San Francisco: Morgan Kaufmann, 1996.

[4] M. Fischer and N. Lynch, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM.*

[5] D. Ongaro and J. Ousterhout, "The Raft Consensus Algorithm," *Lecture Notes CS,* vol. 190, p. 2022, 2015.

[6] L. Lamport, "Paxos Made Simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001),* p. 51–58, 2001.

[7] H. Ng, S. Haridi and P. Carbone, "Omni-Paxos: Breaking the Barriers of Partial Connectivity," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023.

[8] P. Kumari and P. Kaur, "A Survey of Fault Tolerance in Cloud Computing," *Journal of King Saud University - Computer and Information Sciences,* vol. 33, no. 10, p. 1159–1176, 2021.

[9] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *asdsa,* vol. asda, no. asdasd, p. asdas, 2008.

[10] "FOAM," [Online]. Available: https://foam.space/publicAssets/FOAM_Whitepaper.pdf.

[11] S. Lonshakov and A. Krupenkin, "Robonomics: platform for integration of cyber physical systems into human economy," 2018.

[12] "Blockchain and Web3 Strategy --- digital-strategy.ec.europa.eu," [Online]. Available: https://digital-strategy.ec.europa.eu/en/policies/blockchain-strategy.

[13] S. Wong, J. K.-W. Yeung, Y.-Y. Lau and T. Kawasaki, "A Case Study of How Maersk Adopts Cloud-Based Blockchain Integrated with Machine Learning for Sustainable Practices," *Sustainability,* vol. 15, no. 9, p. 7305, 2023.

[14] H. Nguyen and L. Do, "The Adoption of Blockchain in Food Retail Supply Chain: Case: IBM Food Trust Blockchain and the Food Retail Supply Chain in Malta," Lahti University of Applied Sciences, 2018.

[15] E. European Telecommunications Standards Institute, "Intelligent Transport Systems (ITS); Security; Threat, Vulnerability and Risk Analysis (TVRA)," ETSI, Sophia Antipolis Cedex, 2017.

[16] D. C. Gandolfo, L. R. Salinas, M. E. Serrano and J. M. Toibero, "Energy Evaluation of Low-Level Control in UAVs Powered by Lithium Polymer Battery," *ISA Transactions,* vol. 71, p. 563–572, 2017.

[17] M. Podhradský, J. Bone, A. Jensen and C. Coopmans, "Small Low-Cost Unmanned Aerial Vehicle Lithium-Polymer Battery Monitoring System," in *ASME 2013 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2013.

[18] S. Jung, Y. Jo and Y.-J. Kim, "Flight Time Estimation for Continuous Surveillance Missions Using a Multirotor UAV," *Energies,* vol. 12, no. 5, p. 867, 2019.

[19] J. Hu, H. Niu, J. Carrasco, B. Lennox and F. Arvin, "Fault-Tolerant Cooperative Navigation of Networked UAV Swarms for Forest Fire Monitoring," *Aerospace Science and Technology,* vol. 123, p. 107494, 2022.

[20] D. Yu, H. Wu, Y. Sun, L. Zhang and M. Imran, "Adaptive protocol of raft in wireless network," *Ad Hoc Networks,* vol. 154, p. 103377, 2024.

[21] M. Saadoon, S. H. Ab Hamid, H. Sofian, H. H. Altarturi, Z. H. Azizul and N. N. Mohd Daud, "Fault Tolerance in Big Data Storage and Processing Systems: A Review on Challenges and Solutions," *Ain Shams Engineering Journal,* vol. 13, no. 2, p. 101538, 2021.

[22] . Microsoft, "Transactional Outbox pattern with Azure Cosmos DB," 2021. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/best-practices/transactional-outbox-cosmos.

[23] S. Bellovin and W. Cheswick, "Network firewalls," *IEEE communications magazine.*

[24] P. Deutsch, *Falacies of distributed computing.*

[25] L. Lamport, M. Pease and R. Shostak, "The Byzantine generals problem".

[26] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM.*

[27] E. A. Akkoyunlu, K. Ekanadham and R. V. Huber, "Some constraints and tradeoffs in the design of network communications," in *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, 1975.

[28] L. Lamport, "The Part-Time Parliament," in *Concurrency: the Works of Leslie Lamport*, 2019, p. 277–317.

[29] N. C. Strole, "The IBM Token-Ring Network—A Functional Overview," *IEEE Network,* vol. 1, no. 1, p. 23–30, 1987.

[30] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Transactions on Computers,* vol. 31, no. 01, pp. 48-59, 1982.

[31] T. Feng, W. Fan, J. Tang and W. Zeng, "Consensus-based robust clustering and leader election algorithm for homogeneous UAV clusters," in *Journal of Physics: Conference Series*, 2019.

[32] Y. Zou, L. Yang, G. Jing, R. Zhang, Z. Xie, H. Li and D. Yu, "A survey of fault tolerant consensus in wireless networks," *High-Confidence Computing,* vol. 4, no. 2, p. 100202, 2024.

[33] S. Mousavi, F. Afghah, J. D. Ashdown and K. Turck, "Leader-follower based coalition formation in large-scale UAV networks, a quantum evolutionary approach," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2018.

[34] R. Ganesan, X. M. Raajini, A. Nayyar, P. Sanjeevikumar, E. Hossain and A. H. Ertas, "BOLD: Bio-inspired optimized leader election for multiple drones," *Sensors (Basel, Switzerland),* vol. 20, no. 11, p. 3134, 2020.

[35] G. Wang, B.-S. Lee, J. Y. Ahn and G. Cho, "A UAV-Aided Cluster Head Election Framework and Applying Such to Security-Driven Cluster Head Election Schemes: A Survey," *Security and Communication Networks,* vol. 2018, no. 1, p. 6475927, 2018.

[36] M. Mozaffari, W. Saad, M. Bennis, Y.-H. Nam and M. Debbah, "A Tutorial on UAVs for Wireless Networks: Applications, Challenges, and Open Problems," *IEEE Communications Surveys Tutorials,* vol. 21, no. 3, p. 2334–2360, 2019.

[37] I. Jawhar, N. Mohamed, J. Al-Jaroodi, D. P. Agrawal and S. Zhang, "Communication and Networking of UAV-based Systems: Classification and Associated Architectures," *Journal of Network and Computer Applications,* vol. 84, p. 93–108, 2017.

[38] L. Gupta, R. Jain and G. Vaszkun, "Survey of important issues in uav communication," *IEEE Communications Surveys Tutorials,* 2016.

[39] G. Halkes and J. Pouwelse, "UDP NAT and Firewall Puncturing in the Wild," in *NETWORKING 2011*, Berlin, Heidelberg, Springer Berlin Heidelberg, 2011, p. 1–12.

[40] T. Shima and S. Rasmussen, UAV cooperative decision and control: challenges and practical approaches, SIAM, 2009.

[41] V. Khorikov, Unit testing principles, practices, and patterns, Simon and Schuster, 2020.

[42] N. Hao, H. Yi, C. Tian, H. Yao and F. He, "A Distributed-Centralized Dynamic Task Allocation Algorithm for UAVs Tracking Moving Targets," in *2021 40th Chinese Control Conference (CCC)*, 2021.

[43] H. C. Baykara, E. Bıyık, G. Gül, D. Onural, A. S. Öztürk and I. Yıldız, "Real-time detection, tracking and classification of multiple moving objects in UAV videos," in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2017.

[44] F. Cagatay Akyon, E. Akagunduz, S. Onur Altinuc and A. Temizel, "Sequence Models for Drone vs Bird Classification," *arXiv e-prints,* pp. arXiv-2207, 2022.

[45] B. R. Haverkort, Performance of Computer Communication Systems: A Model-Based Approach, Chichester: John Wiley and Sons, 1998.

[46] L. Tseng, "Voting in the Presence of Byzantine Faults," in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2017.

[47] K. Driscoll, B. Hall, M. Paulitsch, P. Zumsteg and H. Sivencrona, "The Real Byzantine Generals," in *23rd Digital Avionics Systems Conference (DASC)*, 2004.

[48] R. G. Engoulou, M. Bellaïche, S. Pierre and A. Quintero, "VANET Security Surveys," *Computer Communications,* vol. 44, pp. 1-13, 2014.

[49] V. Lozupone, "Analyze Encryption and Public Key Infrastructure (PKI)," *International Journal of Information Management,* vol. 38, no. 1, pp. 42-44, 2018.

[50] G. Bracha and S. Toueg, "Asynchronous Consensus and Broadcast Protocols," *Journal of the ACM,* vol. 32, no. 4, pp. 824-840, 1985.

[51] Y. Yeh, "Safety Critical Avionics for the 777 Primary Flight Controls System," in *20th Digital Avionics Systems Conference (DASC)*, 2001.

[52] A. Dahiya and R. Chauhan, "A comparative study of MANET and VANET environment," *Journal of computing,* vol. 2, no. 7, pp. 87-92, 2010.

[53] H. Zhou, W. Xu, J. Chen and W. Wang, "Evolutionary V2X Technologies Toward the Internet of Vehicles: Challenges and Opportunities," *Proceedings of the IEEE,* vol. 108, no. 2, p. 308–323, 2020.

[54] . Statista, "Europe: Passenger Car Parc 2018," 2018. [Online]. Available: https://www.statista.com/statistics/452449/european-countries-number-of-registered-passenger-cars/.

[55] S.-C. Wang, Y.-J. Lin and K.-Q. Yan, "Reaching Byzantine Agreement Underlying VANET," *KSII Transactions on Internet and Information Systems,* vol. 13, no. 7, p. 3351–3368, 2019.

[56] L. Li, J. Liu, L. Cheng, S. Qiu, W. Wang, X. Zhang and Z. Zhang, "CreditCoin: A Privacy-Preserving Blockchain-Based Incentive Announcement Network for Communications of Smart Vehicles," *IEEE Transactions on Intelligent Transportation Systems,* vol. 19, no. 7, p. 2204–2220, 2018.

[57] H. Liu, C.-W. Lin, E. Kang, S. Shiraishi and D. M. Blough, "A Byzantine-Tolerant Distributed Consensus Algorithm for Connected Vehicles Using Proof-of-Eligibility," in *22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, 2019.

[58] Z. Cao, J. Kong, U. Lee, M. Gerla and Z. Chen, "Proof-of-Relevance: Filtering False Data via Authentic Consensus in Vehicle Ad-hoc Networks," in *IEEE INFOCOM Workshops*, 2008.

[59] A. Nataraju, H. D. Maheshappa and A. Devkatte, "Performance Analysis of HWMP Protocol for Wireless Mesh Networks using NS3," in *2016 IEEE Region 10 Conference (TENCON)*, 2016.

[60] D. Rjazanovs and E. Petersons, "Decentralized Byzantine Fault Tolerant Proof of Location," in *PoEM Workshops*, 2020.

[61] Z. Yu, Y. Zhang, B. Jiang, J. Fu and Y. Jin, "A Review on Fault-Tolerant Cooperative Control of Multiple Unmanned Aerial Vehicles," *Chinese Journal of Aeronautics,* vol. 35, no. 1, p. 1–18, 2022.

APPENDIX LIST

Appendix 1

List of publications in full-text conference proceedings and scientific journals.

Appendix 2

NS-3 simulation code for the broadcast algorithm.

Appendix 3

Some components code of the real-time UAV simulation.

Appendix 4

Buzen's algorithm implementation in Python

Appendix 5

Response time distribution and lost monitoring time calsulation implementation in Python.

Appendix 1

List of publications

1. **Rjazanovs D**., Pētersons E., Ipatovs A., Juškaite L., Yeryomin R. "Byzantine Failures and Vehicular Networks." // 2021 IEEE Microwave Theory and Techniques in Wireless Communications (MTTW). Latvia, Riga, 2021. pp. 30–34.
2. **Rjazanovs D**., Petersons E. "Decentralized Byzantine Fault-Tolerant Proof of Location." // PoEM Workshops, 2020. pp. 1–10.
3. **Rjazanovs D**., Ratkuns A., Kārkliņš T., Nagla I., Ipatovs A. "Making Ping Feint to Avoid Service State Desynchronization." // 2023 Workshop on Microwave Theory and Technology in Wireless Communications (MTTW). 2023. pp. 34–38.
4. **Rjazanovs D**., Grabs E., Pētersons E., Aleksandrovs-Moisejs D., Chen T., Čulkovs D., Aleksandrovs M., Ipatovs A. "Drone Cooperation, NAT Traversal, and Performance." // 2024 Photonics & Electromagnetics Research Symposium (PIERS). 2024. pp. 1–6.
5. **Rjazanovs D**., Grabs E., Pētersons E., Lahs A., Kopats A., Kārkliņš T., Aleksandrovs-Moisejs D., Titovics J., Chen T., Čulkovs D., Aleksandrovs M., Ipatovs A. "Analysis of Leader Election Algorithms in the Context of Cooperative UAV Mission Simulation." // 2025 Photonics & Electromagnetics Research Symposium (PIERS). 2025.
6. Aleksandrovs-Moisejs D., Ipatovs A., Grabs E., **Rjazanovs D**. "Evaluation of a Long-Distance IEEE 802.11 ah Wireless Technology in Linux Using Docker Containers." // Elektronika ir Elektrotechnika, Vol. 28, No. 3, 2022, pp. 71–77.
7. Chen T., Grabs E., Petersons E., Efrosinin D., Ipatovs A., Bogdanovs N., **Rjazanovs D**. "Multiclass Live Streaming Video Quality Classification Based on Convolutional Neural Networks." // Automatic Control and Computer Sciences, Vol. 56, No. 5, 2022, pp. 455–466.
8. Aleksandrovs-Moisejs D., Ipatovs A., Grabs E., **Rjazanovs D**., Sinuks I. "Arduino-based Temperature Sensor Organization and Design." // 2023 PhotonIcs & Electromagnetics Research Symposium (PIERS). 2023. pp. 1408–1415.
9. Andrejevs D., Grabs E., Čulkovs D., Jeralovičs V., Juškaite L., Chen T., **Rjazanovs D**., Ipatovs A. "Development of Laser Communication Algorithm for Moving Objects." // 2024 Photonics & Electromagnetics Research Symposium (PIERS). 2024. pp. 1–4.
10. Klūga J., Grabs E., Chen T., Ancāns A., Stetjuha M., **Rjazanovs D**., Ipatovs A. "Motion Sensors Data Fusion for Accurate Measurement in AHRS Systems." // 2024 Photonics & Electromagnetics Research Symposium (PIERS). 2024. pp. 1–4.

NS-3 simulation code for the broadcast algorithm

```
#include "ns3/applications-module.h"
#include "ns3/core-module.h"
#include "ns3/internet-module.h"
#include "ns3/mesh-helper.h"
#include "ns3/mesh-module.h"
#include "ns3/mobility-module.h"
#include "ns3/network-module.h"
#include "ns3/udp-socket-factory.h"
#include "ns3/socket.h"
#include "ns3/inet-socket-address.h"
#include "ns3/yans-wifi-helper.h"
#include "ns3/random-variable-stream.h"
#include <cstring>
#include <iostream>
#include <sstream>
#include <ctime>
#include <vector>
#include <cstdlib> // for std::rand()

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("BroadcastMeshExample");

class MeshTest
{
public:
MeshTest();
void Configure(int argc, char **argv);
int Run();

private:
int m_xSize;
int m_ySize;
int bRetryCount;
double maxSDelay;
double m_step;
double m_randomStart;
double m_totalTime;
uint16_t m_broadcastPort;
uint32_t m_nIfaces;
bool m_chan;
bool m_pcap;
bool m_ascii;
std::string m_stack;
std::string m_root;
uint32_t m_packetSize;
NodeContainer nodes;
NetDeviceContainer meshDevices;
Ipv4InterfaceContainer interfaces;
MeshHelper mesh;
```

```
std::vector< std::vector<uint32_t> > m_receivedIds;

Time m_minBroadcastSendTime;
Time m_maxReceptionTime;

void CreateNodes();
void InstallInternetStack();
void InstallApplication();
void SendBroadcast(Ptr<Socket> socket);
void ReceivePacket(Ptr<Socket> socket);
void LogReceivedIds();
};

MeshTest::MeshTest()
: m_xSize(3),
m_ySize(3),
m_step(50.0),
m_randomStart(0.1),
m_totalTime(10.0),
m_broadcastPort(9),
m_nIfaces(1),
m_chan(true),
m_pcap(false),
m_ascii(false),
m_stack("ns3::Dot11sStack"),
m_root("ff:ff:ff:ff:ff:ff"),
m_packetSize(1000),
m_minBroadcastSendTime(Seconds(1e9)), // initialize to a very large time
m_maxReceptionTime(Seconds(0))
{
}

void
MeshTest::Configure(int argc, char **argv)
{
CommandLine cmd(__FILE__);
cmd.AddValue("x-size", "Number of nodes in a row grid", m_xSize);
cmd.AddValue("y-size", "Number of rows in a grid", m_ySize);
cmd.AddValue("step", "Size of edge in our grid (meters)", m_step);
cmd.AddValue("start", "Maximum random start delay for beacon jitter (sec)", m_randomStart);
cmd.AddValue("time", "Simulation time (sec)", m_totalTime);
cmd.AddValue("port", "Broadcast port", m_broadcastPort);
cmd.AddValue("interfaces", "Number of radio interfaces used by each mesh point", m_nIfaces);
cmd.AddValue("channels", "Use different frequency channels for different interfaces", m_chan);
cmd.AddValue("pcap", "Enable PCAP traces on interfaces", m_pcap);
cmd.AddValue("ascii", "Enable Ascii traces on interfaces", m_ascii);
cmd.AddValue("stack", "Type of protocol stack. ns3::Dot11sStack by default", m_stack);
cmd.AddValue("root", "Mac address of root mesh point in HWMP", m_root);
cmd.AddValue("maxSDelay", "Maximum start delay for broadcast jitter", maxSDelay);
cmd.AddValue("bRetryCount", "Number of broadcast retries", bRetryCount);
cmd.AddValue("packetSize", "Packet payload size in bytes", m_packetSize);
cmd.Parse(argc, argv);
```

```
NS_LOG_DEBUG("Grid: " << m_xSize << " x " << m_ySize);
NS_LOG_DEBUG("Simulation time: " << m_totalTime << " s");
if (m_ascii)
{
PacketMetadata::Enable();
}
}

void
MeshTest::CreateNodes()
{
nodes.Create(m_ySize * m_xSize);

m_receivedIds.resize(nodes.GetN());

YansWifiPhyHelper wifiPhy;
YansWifiChannelHelper wifiChannel = YansWifiChannelHelper::Default();
wifiPhy.SetChannel(wifiChannel.Create());
mesh = MeshHelper::Default();
if (!Mac48Address(m_root.c_str()).IsBroadcast())
{
mesh.SetStackInstaller(m_stack, "Root", Mac48AddressValue(Mac48Address(m_root.c_str())));
}
else
{
mesh.SetStackInstaller(m_stack);
}
if (m_chan)
{
mesh.SetSpreadInterfaceChannels(MeshHelper::SPREAD_CHANNELS);
}
else
{
mesh.SetSpreadInterfaceChannels(MeshHelper::ZERO_CHANNEL);
}
mesh.SetMacType("RandomStart", TimeValue(Seconds(m_randomStart)));
mesh.SetNumberOfInterfaces(m_nIfaces);

meshDevices = mesh.Install(wifiPhy, nodes);
mesh.AssignStreams(meshDevices, 0);

MobilityHelper mobility;
mobility.SetPositionAllocator("ns3::GridPositionAllocator",
"MinX", DoubleValue(0.0),
"MinY", DoubleValue(0.0),
"DeltaX", DoubleValue(m_step),
"DeltaY", DoubleValue(m_step),
"GridWidth", UintegerValue(m_xSize),
"LayoutType", StringValue("RowFirst"));
mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobility.Install(nodes);

if (m_pcap)
```

```
{
wifiPhy.EnablePcapAll("mp");
}
if (m_ascii)
{
AsciiTraceHelper ascii;
wifiPhy.EnableAsciiAll(ascii.CreateFileStream("mesh.tr"));
}
}

void
MeshTest::InstallInternetStack()
{
InternetStackHelper internetStack;
internetStack.Install(nodes);
Ipv4AddressHelper address;
address.SetBase("10.1.1.0", "255.255.255.0");
interfaces = address.Assign(meshDevices);
}

void
MeshTest::ReceivePacket(Ptr<Socket> socket)
{
Address from;
Ptr<Packet> packet = socket->RecvFrom(from);
while (packet)
{
char buffer[64] = {0};
packet->CopyData((uint8_t*)buffer, sizeof(buffer) - 1);
std::string idStr(buffer);
uint32_t senderId = std::stoi(idStr);

uint32_t nodeId = socket->GetNode()->GetId();
if (senderId != nodeId)
{
if (std::find(m_receivedIds[nodeId].begin(), m_receivedIds[nodeId].end(), senderId) ==
m_receivedIds[nodeId].end())
{
m_receivedIds[nodeId].push_back(senderId);
// NS_LOG_INFO("Node " << nodeId << " received broadcast from node "
// << senderId << " at " << Simulator::Now().GetSeconds() << " s");

if (Simulator::Now() > m_maxReceptionTime)
{
m_maxReceptionTime = Simulator::Now();
}
}
}
packet = socket->RecvFrom(from);
}
}

void
```

```
MeshTest::SendBroadcast(Ptr<Socket> socket)
{
uint32_t nodeId = socket->GetNode()->GetId();
std::string idStr = std::to_string(nodeId);
size_t idLen = idStr.size();

if (m_packetSize < idLen + 1)
{
NS_LOG_WARN("Packet size (" << m_packetSize << ") is too small to hold node ID string (" << idStr <<
")");
return;
}

std::vector<char> payload(m_packetSize, 0);
memcpy(payload.data(), idStr.c_str(), idLen);
payload[idLen] = '\0';
for (size_t i = idLen + 1; i < m_packetSize; i++)
{
payload[i] = static_cast<char>(std::rand() % 256);
}

Ptr<Packet> packet = Create<Packet>((uint8_t*)payload.data(), m_packetSize);
socket->Send(packet);
if (Simulator::Now() < m_minBroadcastSendTime)
{
m_minBroadcastSendTime = Simulator::Now();
}
}

void
MeshTest::InstallApplication()
{
Ptr<UniformRandomVariable> rand = CreateObject<UniformRandomVariable>();
rand->SetAttribute("Min", DoubleValue(0.001));
rand->SetAttribute("Max", DoubleValue(maxSDelay));

for (uint32_t i = 0; i < nodes.GetN(); i++)
{
Ptr<Node> node = nodes.Get(i);
Ptr<Socket> recvSocket = Socket::CreateSocket(node, UdpSocketFactory::GetTypeId());
InetSocketAddress local = InetSocketAddress(Ipv4Address::GetAny(), m_broadcastPort);
recvSocket->Bind(local);
recvSocket->SetRecvCallback(MakeCallback(&MeshTest::ReceivePacket, this));
}

for (uint32_t i = 0; i < nodes.GetN(); i++)
{
Ptr<Node> node = nodes.Get(i);
Ptr<Socket> sendSocket = Socket::CreateSocket(node, UdpSocketFactory::GetTypeId());
sendSocket->SetAllowBroadcast(true);
InetSocketAddress remote = InetSocketAddress(Ipv4Address("10.1.1.255"), m_broadcastPort);
sendSocket->Connect(remote);
```

```
for (int i = 0; i < bRetryCount; i++)
{
double delay = rand->GetValue();
Simulator::Schedule(Seconds(1) + Seconds(delay), &MeshTest::SendBroadcast, this, sendSocket);
}
}
}

void
MeshTest::LogReceivedIds()
{
std::cout << "\n--- Received IDs per node ---" << std::endl;
for (uint32_t i = 0; i < m_receivedIds.size(); i++)
{
std::cout << "Node " << i << ": ";
for (auto id : m_receivedIds[i])
{
std::cout << id << " ";
}
std::cout << std::endl;
}
}

int
MeshTest::Run()
{
CreateNodes();
InstallInternetStack();
InstallApplication();
Simulator::Stop(Seconds(m_totalTime));
Simulator::Run();
LogReceivedIds();

bool allReceived = true;
for (uint32_t i = 0; i < nodes.GetN(); i++)
{
if (m_receivedIds[i].size() != nodes.GetN() - 1)
{
allReceived = false;
break;
}
}
if (allReceived)
{
double floodingTimeMs = (m_maxReceptionTime - m_minBroadcastSendTime).GetMilliSeconds();
Simulator::Destroy();
// std::cout << "Flooding took " << floodingTimeMs << " ms." << std::endl;
return floodingTimeMs;
}
else
{
Simulator::Destroy();
// std::cout << "Not all nodes received all broadcasts." << std::endl;
return 0;
```

```
}
}

int
main(int argc, char **argv)
{
int simCount = std::atoi(argv[argc - 1]);
int64_t totalTime = 0;
int failed = 0;
for (int i = 0; i < simCount; i++)
{
uint32_t seed = static_cast<uint32_t>(time(nullptr));
RngSeedManager::SetSeed(seed);
RngSeedManager::SetRun(1);
LogComponentEnable("BroadcastMeshExample", LOG_LEVEL_INFO);
MeshTest t;
t.Configure(argc, argv);
int runTime = t.Run();
if (runTime == 0)
{
failed++;
}
totalTime += runTime;
}
int averageTime = totalTime / (simCount - failed);
std::cout << "Average simulation time over " << (simCount - failed)
<< " runs: " << averageTime << " ms" << std::endl;
return 0;

// Example run:
// ./ns3 run "scratch/mymesh.cc --x-size=6 --y-size=2 --maxSDelay=0.3 --step=35 --bRetryCount=1 --
packetSize=1000 50"
}
```

Some components code of the real-time UAV simulation

```
using chat.Middleware.FailureDetector.Perfect;
using Serilog;

namespace chat.Middleware.FailureDetector.Eventual;

public class EventualFailureDetectorCore
{
private readonly string[] nodeIds;
private int timeout;
private readonly int timeoutDelta;
private readonly List<string> lastCycleNodeAcks;
private readonly List<string> stillSuspected;

public EventualFailureDetectorCore(string[] nodeIds, int timeout, int timeoutDelta)
{
this.nodeIds = nodeIds;
this.timeout = timeout;
this.timeoutDelta = timeoutDelta;
this.lastCycleNodeAcks = [.. nodeIds];
this.stillSuspected = [];
}

public void AckReceived(string nodeId)
{
lock (this)
{
lastCycleNodeAcks.Add(nodeId);
}
}

public EventualDetectorDecision PrepareNodesForHeartBeats()
{
lock (this)
{
var atLeastOneRestore = false;
var suspectedToReport = new List<string>();
var restoredToReport = new List<string>();
foreach (var nodeId in nodeIds)
{
if (!lastCycleNodeAcks.Contains(nodeId))
{
if (!stillSuspected.Contains(nodeId))
{
suspectedToReport.Add(nodeId);
stillSuspected.Add(nodeId);
Log.Information($"$$$ Suspected {nodeId}");
}

}
else
```

```
{
if (stillSuspected.Contains(nodeId))
{
restoredToReport.Add(nodeId);
stillSuspected.Remove(nodeId);

if (atLeastOneRestore == false)
{
timeout += timeoutDelta;
atLeastOneRestore=true;
Log.Information($"$$$ Timeout increased to {timeout}");
}

}
}

}
lastCycleNodeAcks.Clear();
return new EventualDetectorDecision(
nodeIds.ToList(),
suspectedToReport,
restoredToReport,
timeout);
}
}
}

public record EventualDetectorDecision(
List<string> NodesToSendHeartbeat,
List<string> NodesToReportSuspect,
List<string> NodesToReportRestore,
int Timeout
);



using System.Collections.Concurrent;
using Serilog;

namespace chat.Middleware.FailureDetector.Perfect;

public class TimeoutFailureDetectorCore
{
private readonly ConcurrentDictionary<string, string> aliveNodes;
private readonly ConcurrentDictionary<string, string> lastCycleAcks;

public TimeoutFailureDetectorCore(string[] nodeIds)
{
this.aliveNodes = new(nodeIds.Select(x => new KeyValuePair<string, string>(x, x)));
this.lastCycleAcks = new(nodeIds.Select(x => new KeyValuePair<string, string>(x, x)));
}

public void AckReceived(string node)
```

```
{
lock (this)
{
if (!aliveNodes.TryGetValue(node, out var _))
return;

lastCycleAcks.TryAdd(node, node);
}
}

public FailureDetectorDecision PrepareNodesForHeartBeats()
{
lock (this)
{
var crashed = new List<string>();
var alive = new List<string>();
foreach (var node in aliveNodes.Values)
{
if (lastCycleAcks.Values.Contains(node))
{
alive.Add(node);
continue;
}

crashed.Add(node);
aliveNodes.TryRemove(new(node,node));
}
lastCycleAcks.Clear();
return new FailureDetectorDecision(alive, crashed);
}
}
}

using System.Collections.Concurrent;
using System.Net;
using System.Net.Sockets;
using System.Text;
using chat.Messenger;
using chat.Messenger.Config;
using chat.Messenger.PerfectLink;
using chat.Proxy.Behaviors;
using Serilog;

namespace chat.Proxy;

public class UdpProxy : IDisposable
{
private readonly IPacketDropBehavior _dropBehavior;
private readonly IPacketDelayBehavior _delayBehavior;
private readonly int _workerCount;
private UdpClient _mainUdpClient;
private CancellationTokenSource _cancelToken;
```

```
private readonly ConcurrentBag<Timer> _timers = new ConcurrentBag<Timer>(); // Store timers to
prevent GC



public UdpProxy(UdpClient udpClient, IPacketDropBehavior dropBehavior, IPacketDelayBehavior
delayBehavior, int wORKER_COUNT)
{
_dropBehavior = dropBehavior;
_delayBehavior = delayBehavior;
_workerCount = wORKER_COUNT;
_mainUdpClient = udpClient;
_cancelToken = new CancellationTokenSource();
}

public void Start()
{
// BlockingCollection is used to handle the queue of incoming messages.
BlockingCollection<(byte[], IPEndPoint)> packetQueue = new BlockingCollection<(byte[], IPEndPoint)>();

// Start a dedicated task for receiving packets asynchronously.
Task.Run(async () =>
{
while (!_cancelToken.IsCancellationRequested)
{
try
{
UdpReceiveResult receiveResult = await _mainUdpClient.ReceiveAsync();
if (!_dropBehavior.ToBeDropped())
{
packetQueue.Add((receiveResult.Buffer, receiveResult.RemoteEndPoint));
}
else
{
Log.Verbose("Dropping {@buffer}", Encoding.UTF8.GetString(receiveResult.Buffer));
}
}
catch (Exception ex)
{
Log.Error(ex, ex.Message);
}
}
}, _cancelToken.Token);

Task[] workers = new Task[_workerCount];

var activeWcount = 0;
for (int i = 0; i < _workerCount; i++)
{
workers[i] = Task.Run(() =>
{
activeWcount++;
```

```
foreach (var (receivedBytes, senderEndPoint) in
packetQueue.GetConsumingEnumerable(_cancelToken.Token))
{
string forwardIP = "", forwardPort;
try
{
string receivedMessage = Encoding.UTF8.GetString(receivedBytes);
var sourceIP = senderEndPoint.Address.ToString();
var sourcePort = senderEndPoint.Port;
string dataToSend;
ParseMessage(receivedMessage, out forwardIP, out forwardPort, out dataToSend);
var delay = _delayBehavior.NextDelayMs();

// Use Timer to delay forwarding
_timers.Add(new Timer(state =>
{
try
{
// Log.Verbose($"Packet delayed for {delay}ms");
var msgWithOriginalSrc = $"{dataToSend};;;{sourceIP}:{sourcePort}";
var bytes = Encoding.UTF8.GetBytes(msgWithOriginalSrc);
_mainUdpClient.Send(bytes, bytes.Length, forwardIP, int.Parse(forwardPort));
// Log.Verbose("Delayed for {@delay}ms. Forwarded to {@forwardIP}:{@forwardPort}. Msg:
'{@dataToSend}'",delay, forwardIP, forwardPort, dataToSend);
}
catch (Exception ex)
{
Log.Error(ex, $"{ex.Message}. Additional info: forwardIp '{forwardIP}'");
}
}, null, delay, Timeout.Infinite));

// The timer will trigger once after the specified delay time and send the packet
}
catch (Exception ex)
{
Log.Error(ex, $"{ex.Message}. Additional info: forwardIp '{forwardIP}'");
}
}
});

}

// Ensure the cancellation token is handled properly.
Task.WhenAll(workers).ContinueWith(t =>
{
packetQueue.Dispose();
});
Task.Run(() =>
{
while (true)
{
if (activeWcount > _workerCount - 1)
break;
}
```

```
}).Wait();
}


private static void ParseMessage(string receivedMessage, out string forwardIP, out string forwardPort,
out string dataToSend)
{
var msgParts = receivedMessage.Split(";;;");
if (msgParts.Count() < 2)
throw new Exception("No delimiter delimiter ';;;' before forward address.");

var forwardMessageParts = msgParts[1].Split(":");
if (forwardMessageParts.Count() < 2)
throw new Exception($"Wrong forward IP: '{msgParts[1]}'");

forwardIP = forwardMessageParts[0];
forwardPort = forwardMessageParts[1];
dataToSend = msgParts[0];
}

public void Dispose()
{
_cancelToken.Cancel();
_mainUdpClient.Dispose();
}
}


using System.Text.Json;
using chat.Domain.Models;
using chat.Messenger.Broadcast;
using chat.Messenger.ExtensionMethods;
using chat.Messenger.MessageTypes;
using chat.Messenger.ValueTypes;
using Serilog;

namespace chat.Middleware.Consensus;


public class FailStopFloodingConsensus<TConsensusValue> where TConsensusValue : class
{
private readonly BestEffortBroadcast _broadcast;

private ConsensusStateMachine<TConsensusValue> _consensus { get; init; }

public FailStopFloodingConsensus(
BestEffortBroadcast broadcast,
NetworkAddress me,
Func<IEnumerable<TConsensusValue>, TConsensusValue> decider,
IEnumerable<NetworkAddress> allNodes
)
{
var allNeighbors = allNodes.Where(x=>x.IPAddress != me.IPAddress && x.Port != me.Port);
```

```
_broadcast = broadcast;
_consensus = new ConsensusStateMachine<TConsensusValue>(me, decider,
typeof(TConsensusValue).ToString(), allNeighbors);
ListenForConsensusMessages();
}

public void HandleCrash(NetworkAddress failedNode)
{
_consensus.NodeCrashed(failedNode);
}

public async Task<TConsensusValue> Propose(TConsensusValue proposedValue,
Action<TConsensusValue>? commitFunc = null)
{
try
{
_consensus.Init(proposedValue);
RequestMessageAndBroadcast();
while (true)
{
RequestMessageAndBroadcast();

if (_consensus.Decision is not null)
break;

await Task.Delay(50);
}
RequestMessageAndBroadcast();
return _consensus.Decision;
}
finally
{
_consensus.Finish();
if (commitFunc is not null)
{
commitFunc(_consensus.Decision);
}
}
}

private void RequestMessageAndBroadcast()
{
var msg = _consensus.RequestProposalMsg();
if (msg is null)
return;

Broadcast(msg.Nodes, msg.ConsensusProposalDTO);
}

private void Broadcast(IEnumerable<NetworkAddress> nodes,
ConsensusProposalDTO<TConsensusValue> consensusProposalDTO)
{
var msgId = Guid.NewGuid();
```

```
var msg = new BebBroadcastDTO
{
MessageType = "BebBroadcastDTO",
MsgId = msgId,
Text = JsonSerializer.Serialize(consensusProposalDTO),
};
_broadcast.BroadcastBestEffortWithAck(nodes, msg, msgId);
}

private void ListenForConsensusMessages()
{
_broadcast.SubscribeToBebDeliver((m, ctx) =>
{
var msgType = TryGetMessageType(m.Text);
if (msgType == "ConsensusProposalDTO")
{
var proposal = JsonSerializer.Deserialize<ConsensusProposalDTO<TConsensusValue>>(m.Text);
if (proposal is null)
{
return;
}
Task.Run(async () =>
{
while (_consensus.ConsensusId != proposal.Id)
{
await Task.Delay(1000);
}
_consensus.ReceiveMessage(proposal, ctx.ToNetworkAddress());
});
}
});
}


private string TryGetMessageType(string msg)
{
try
{
return JsonSerializer.Deserialize<MessageTypeDTO>(msg)?.MessageType;
}
catch (System.Exception)
{
return null;
}
}

}


using chat.Domain.InternalEvents;
using chat.Domain.Messages;
using chat.Domain.Models;
using chat.Messenger;
```

```
using chat.Messenger.ValueTypes;
using chat.Middleware;
using chat.Middleware.AddressResolution;
using chat.Middleware.DTOs.Chatting;
using chat.Middleware.FailureDetector;
using chat.Middleware.FailureDetector.Perfect;
using chat.SimulatedDrone.DB;
using chat.SimulatedDrone.PhysicalTasks;
using chat.SimulatedDrone.UseCases;
using chat.SimulatedDrone.UseCases.Centralized;
using chat.SimulatedDrone.UseCases.DynamicLeader;
using Serilog;

namespace chat.SimulatedDrone.EventRouting;

public class DynamicLeaderEventRouter : IDisposable
{
private readonly IAddressResolver _addressResolver;
private readonly IMiddleware _middleware;
private readonly string _droneId;
private readonly PatrolUseCase _patrolUseCase;
private readonly UAVCrashedUseCase _uAVCrashedUseCase;
private readonly LocationUpdateUseCase _locationUpdateUseCase;
private readonly ThreatInTheViewEventUseCase _threatInTheViewEventUseCase;
private readonly ThreatControlUseCase _threatControlUseCase;
private readonly ThreatGoneFromViewEventUseCase _threatGoneFromViewEventUseCase;
private readonly UAVRestoredUseCase uAVRestoredUseCase;
private readonly EventualFailDetector _eventualFailDetector;
private TimeoutFailDetector _timeoutFailureDetector;
private object _lock = new();

public DynamicLeaderEventRouter(
IAddressResolver addressResolver,
VisualController visualController,
IMiddleware middleware,
string droneId,
PatrolUseCase patrolUseCase,
UAVCrashedUseCase uAVCrashedUseCase,
LocationUpdateUseCase locationUpdateUseCase,
ThreatInTheViewEventUseCase threatInTheViewEventUseCase,
ThreatControlUseCase threatControlUseCase,
ThreatGoneFromViewEventUseCase threatGoneFromViewEventUseCase,
StateRepository stateRepository,
IReliableLink link,
int failDetectorTimeoutMs,
int? deltaTimeoutMs,
UAVRestoredUseCase uAVRestoredUseCase
)
{
_middleware = middleware;
_droneId = droneId;
_patrolUseCase = patrolUseCase;
_uAVCrashedUseCase = uAVCrashedUseCase;
_locationUpdateUseCase = locationUpdateUseCase;
```

```
_threatInTheViewEventUseCase = threatInTheViewEventUseCase;
_threatControlUseCase = threatControlUseCase;
_threatGoneFromViewEventUseCase = threatGoneFromViewEventUseCase;
this.uAVRestoredUseCase = uAVRestoredUseCase;
_addressResolver = addressResolver;
visualController.SubscribeOnThreatGoneFromViewEvent(HandleThreatGoneFromViewEvent);
visualController.SubscribeOnThreatInTheViewEvent(HandleThreatInTheViewEvent);
_middleware.SubscribeToPrivateMsgDeliver(HandleIncomingMessage);


var droneIds = stateRepository.GetAllExistingDrones().Where(x => x != droneId).ToArray();
List<NetworkAddress> droneNetworkAddresses = BaseNode.ConvertIdsToNetworkAddresses(droneIds,
addressResolver);
if (deltaTimeoutMs is null)
{
_timeoutFailureDetector = new TimeoutFailDetector(droneNetworkAddresses, link,
failDetectorTimeoutMs);
_timeoutFailureDetector.Subscribe(HandlerCrashDetected);
}
else
{
_eventualFailDetector = new EventualFailDetector(droneNetworkAddresses, link, failDetectorTimeoutMs,
HandlerCrashDetected, HandleRestore, deltaTimeoutMs.Value);
}

}

private void HandleRestore(NetworkAddress address)
{
var username = _addressResolver.GetOrCreateAddressDomain(address.IPAddress.ToString(),
address.Port.ToInt(), null);
uAVRestoredUseCase.Invoke(username, _lock);
}


private void HandleThreatGoneFromViewEvent(ThreatGoneFromViewEvent eventt)
{
_threatGoneFromViewEventUseCase.Invoke(eventt, _droneId, _lock);
}

private void HandleThreatInTheViewEvent(ThreatInTheViewEvent eventt)
{
_threatInTheViewEventUseCase.Invoke(eventt,_lock);
}

private void HandlerCrashDetected(NetworkAddress address)
{
var username = _addressResolver.GetOrCreateAddressDomain(address.IPAddress.ToString(),
address.Port.ToInt(), null);
_uAVCrashedUseCase.Invoke(username,_lock);
}

private void HandleIncomingMessage(PrivateMiddlewareMessage msg)
```

```
{
try
{

if (NetworkMsg.Type(msg.Text) == typeof(TimestampedCoordinates))
{
var payload = NetworkMsg.Deserialize<TimestampedCoordinates>(msg.Text);
_locationUpdateUseCase.Invoke(payload);
return;
}
Task task = null;
if (NetworkMsg.Type(msg.Text) == typeof(PatrolPath))
{
var payload = NetworkMsg.Deserialize<PatrolPath>(msg.Text);
task = _patrolUseCase.Invoke(payload, msg.Timestamp);
}
if (NetworkMsg.Type(msg.Text) == typeof(ThreatControlTask))
{
var payload = NetworkMsg.Deserialize<ThreatControlTask>(msg.Text);
task = _threatControlUseCase.Invoke(payload, msg.Timestamp);
}
if (NetworkMsg.Type(msg.Text) == typeof(ThreatInTheViewEvent))
{
var payload = NetworkMsg.Deserialize<ThreatInTheViewEvent>(msg.Text);
task = _threatInTheViewEventUseCase.Invoke(payload,_lock);
}
if (NetworkMsg.Type(msg.Text) == typeof(ThreatGoneFromViewEvent))
{
var payload = NetworkMsg.Deserialize<ThreatGoneFromViewEvent>(msg.Text);
task = _threatGoneFromViewEventUseCase.Invoke(payload, msg.Author,_lock);
}
if (task is not null)
{
task.ContinueWith(task =>
{
if (task.IsFaulted)
{
Log.Error(task.Exception?.GetBaseException(), task.Exception?.GetBaseException().Message);
}
}, TaskContinuationOptions.OnlyOnFaulted);
return;
}
Log.Error($"Incoming message type is not supproted: '{msg.Text}'");

}
catch (Exception ex)
{
Log.Error(ex, ex.Message);
}
}

public void Dispose()
{
_timeoutFailureDetector?.Dispose();
```

```
_eventualFailDetector?.Dispose();
}
}
```

```
using chat.Domain;
using chat.Domain.Models;

namespace chat.DroneCore.PureLogic;

public class ThreatControlTaskCalculator
{
public static ThreatControlTask[] AssignDronesToThreats(
IEnumerable<DroneCoord> aliveDrones,
IEnumerable<Threat> activeThreats,
ThreatControlTask[] lastMissionTasks)
{
lastMissionTasks =
lastMissionTasks.Where(x=>activeThreats.Select(x=>x.ThreatId).Contains(x.Threat.ThreatId)).ToArray();
var dronesAlreadyTracking = lastMissionTasks.SelectMany(x => x.DronePositions).Select(x =>
x.DroneId).ToArray();
var notTrackingDrones = aliveDrones.Where(x => !dronesAlreadyTracking.Contains(x.DroneId));
var leftDrones = notTrackingDrones;
string patrolId = null;

if (!activeThreats.Any())
return [];

patrolId = UAVFarestFromAllThreats(notTrackingDrones, activeThreats);
leftDrones = notTrackingDrones.Where(drone => drone.DroneId != patrolId).ToList();

if (!leftDrones.Any())
return lastMissionTasks;

var newTasks = new List<ThreatControlTask>();

var partiallyControlledTasks = lastMissionTasks.Where(x => x.DronePositions.Count() == 1).ToArray();
var fullyControlledTasks = lastMissionTasks.Where(x => x.DronePositions.Count() > 1).ToArray();
var notControlledThreats = activeThreats.Where(x =>
!partiallyControlledTasks.Select(x => x.Threat.ThreatId).Contains(x.ThreatId) &&
!fullyControlledTasks.Select(x => x.Threat.ThreatId).Contains(x.ThreatId));

//first assign not controlled
foreach (var threat in notControlledThreats)
{
if (!leftDrones.Any())
break;

//take closest free drone
```

```
var closestDrone = leftDrones.OrderBy(x => TwoDHelpers.CalculateDistance(x.Coord,
threat.Coordinates)).First();
newTasks.Add(
new ThreatControlTask(threat, [new DroneThreatControl(closestDrone.DroneId,
TrackingControlSide.Up)])
);
leftDrones = leftDrones.Where(x => x.DroneId != closestDrone.DroneId);
}

// then assign partially controlled
List<ThreatControlTask> newFullyControlled = [];
ThreatControlTask[] allPartiallyControlledTasks = [.. partiallyControlledTasks, .. newTasks];
foreach (var task in allPartiallyControlledTasks)
{
if (!leftDrones.Any())
break;

var closestDrone = leftDrones.OrderBy(x => TwoDHelpers.CalculateDistance(x.Coord,
task.Threat.Coordinates)).First();
newFullyControlled.Add(
new ThreatControlTask(task.Threat, [task.DronePositions.First(), new
DroneThreatControl(closestDrone.DroneId, TrackingControlSide.Down)])
);
leftDrones = leftDrones.Where(x => x.DroneId != closestDrone.DroneId);
}
var leftPartiallyControlled = allPartiallyControlledTasks.Where(x => !newFullyControlled.Select(x =>
x.Threat.ThreatId).Contains(x.Threat.ThreatId));

return [.. fullyControlledTasks, .. newFullyControlled, .. leftPartiallyControlled];
}

public static string UAVFarestFromAllThreats(IEnumerable<DroneCoord> drones, IEnumerable<Threat>
threats)
{
var droneList = drones.ToList();
var threatList = threats.ToList();

// Step 1: Calculate the minimum distance to any threat for each drone
var droneDistances = droneList.Select(drone => new
{
Drone = drone,
MinDistanceToAnyThreat = threatList.Min(threat => TwoDHelpers.CalculateDistance(drone.Coord,
threat.Coordinates))
}).ToList();

// Step 2: Find the drone with the largest minimum distance to any threat
var farthestDrone = droneDistances
.OrderByDescending(d => d.MinDistanceToAnyThreat)
.FirstOrDefault();

// Step 3: Return the drone ID if a valid drone is found; otherwise, return null or handle as needed
return farthestDrone?.Drone.DroneId;
}
```

```
}
```

Buzen's algorithm implementation in Python

```
import itertools
import math
import numpy as np

def buzen(K, D):
M=len(D)
G = np.zeros((M, K + 1))
G[:, 0] = 1

for k in range(1, K + 1):
G[0, k] = (D[0] ** k)
for m in range(1, M):
for k in range(1, K + 1):
G[m, k] = G[m - 1, k] + D[m] * G[m, k - 1]
return G

def buzen_inf(K, D):
M=len(D)
G = np.zeros((M, K + 1))
G[:, 0] = 1

for k in range(1, K + 1):
G[0, k] = (D[0] ** k) / math.factorial(k)

for m in range(1, M):
for k in range(1, K + 1):
G[m, k] = G[m - 1, k] + D[m] * G[m, k - 1]

return G

def X(G,K):
return (G[-1,K-1])/(G[-1,K])

def E_Ni(K, D, G,inf_0=False):
M = len(D)
expected_N = np.zeros(M)

for i in range(M):
expected_N[i] = sum(D[i] ** k * G[M-1, K-k] / G[M-1, K] for k in range(1, K + 1))

if inf_0==True:
expected_N[0]=D[0]*(G[-1,K-1])/(G[-1,K])

return expected_N

def E_Ri(K, D, G, V,inf_0=False):
M = len(D)
expected_R = np.zeros(M)
```

```
for i in range(M):
numerator = sum((D[i] ** k) * G[M-1, K-k] for k in range(1, K + 1))
denominator = V[i] * G[M-1, K-1]
expected_R[i] = numerator / denominator

if inf_0==True:
expected_R[0]=D[0]

return expected_R

def util(K, D, G,inf_0=False):
M = len(D)
rho = np.zeros(M)

for i in range(M):
rho[i] = D[i] * G[M-1, K-1] / G[M-1, K]
if inf_0==True:
rho[0]=0

return rho


def mva(K, D, inf_0=False):
N = len(D)
E_N = [0] * N
X_values = []
E_R_values = []

for k in range(1, K + 1):
E_R_i = [(E_N[i] + 1) * D[i] for i in range(N)]

if inf_0:
E_R_i[0] = D[0]

E_R_total = sum(E_R_i)

X_K = k / E_R_total
X_values.append(X_K)

E_N = [X_K * E_R_i[i] for i in range(N)]
E_R_values.append(E_R_i)

rho_values = [D[i] * X_values[-1] for i in range(N)]
if inf_0:
rho_values[0] = 0

return X_values[-1], E_R_values[-1], E_N, rho_values


def compute_state_probability(D, G_MK, state):
```

```
numerator = (D[0] ** state[0] / math.factorial(state[0])) * np.prod([D[i] ** state[i] for i in range(1,
len(D))])
return numerator / G_MK[-1, -1] # Normalize with G(M, K)
```

Response time distribution and lost monitoring time calsulation implementation in Python

```
def compute_response_time_distribution(D, K):
M = len(D)
G = buzen_inf(K, D)
response_time_distribution = {}
probSum = 0

for state in itertools.product(range(K+1), repeat=M):
if sum(state) == K:
raw_response_time = sum((state[i]+1) * D[i] for i in range(1, M)) # Exclude node 0
prob = compute_state_probability(D, G, state)
probSum += prob
response_time_distribution[raw_response_time] = response_time_distribution.get(raw_response_time,
0) + prob

print(f"Sum of all state probabilities before normalization: {probSum}")
return response_time_distribution

def simulate_hybrid_false_positive_loss(response_time_pmf, A0, r, lambda_threat_sec, T_sim):
num_samples=5000
N = int(lambda_threat_sec * T_sim)
n_real = int(N * r)
n_nonthreat = int(N * (1-r))
n_fp = int((1 - A0) * n_nonthreat)
n_fn = int((1 - A0) * n_real)
# print(f"n_fp: {n_fp}")
# print(f"n_fn: {n_fn}")
# print(f"N: {N}")
# print(f"real: {n_real}")



response_times, probabilities = zip(*response_time_pmf.items())
response_times = np.array(response_times)
probabilities = np.array(probabilities)
probabilities /= probabilities.sum()

# Sample directly from discrete PMF for false positives
sampled_response_times_fp = np.random.choice(response_times, size=(n_fp, num_samples),
p=probabilities)
total_wasted_time_fp = np.sum(sampled_response_times_fp, axis=0)

# Sample directly from discrete PMF for false negatives
sampled_response_times_fn = np.random.choice(response_times, size=(n_fn, num_samples),
p=probabilities)
total_wasted_time_fn = np.sum(sampled_response_times_fn, axis=0)

# Total wasted time
total_wasted_time = total_wasted_time_fp + total_wasted_time_fn
```

```
# print(f"total_wasted_time_fp: {total_wasted_time_fp}")
# print(f"total_wasted_time_fn: {total_wasted_time_fn}")
# print(f"total_wasted_time: {total_wasted_time}")

lost_time_fraction = np.clip(1 - total_wasted_time / T_sim, 0, 1)

return lost_time_fraction
```

Dmitrijs Rjazanovs was born in 1994 in Riga. He obtained a Master's degree (2018) in the study program "Intelligent Robotic Systems" from Riga Technical University. He has been working professionally in programming since 2015. Currently, is a software architect at C.T.Co Ltd. and a lecturer at Riga Technical University, teaching the courses "Cryptography" and "Real-Time E-Commerce".
His scientific interests include distributed systems, consensus algorithms, and software engineering.