UNIVERSITY OF THESSALY

DOCTORAL THESIS

System Support for the Fault Tolerance, Testing and Orchestration of Drone Applications

Athanasios (Nasos) GRIGOROPOULOS

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Department of Electrical and Computer Engineering

April 12, 2022

UNIVERSITY OF THESSALY

DOCTORAL THESIS

System Support for the Fault Tolerance, Testing and Orchestration of Drone Applications

Author:
Athanasios (Nasos) GRIGOROPOULOS

Advisor:
Prof. Spyros LALIS

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Department of Electrical and Computer Engineering

Examination Committee:
Spyros Lalis, Professor, University of Thessaly
Christos D. Antonopoulos, Associate Professor, University of Thessaly
Antonios Argyriou, Associate Professor, University of Thessaly
Nikolaos Bellas, Professor, University of Thessaly
Dimitrios Katsaros, Associate Professor, University of Thessaly
Stathes Hadjiefthymiades, Professor, University of Athens
Kostas Magoutis, Associate Professor, University of Crete

April 12, 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Διδακτορικό Δίπλωμα

Υποστήριξη σε επίπεδο συστήματος για την ανοχή βλαβών, δοκιμαστική λειτουργία και ενορχήστρωση σε εφαρμογές με μη επανδρωμένα εναέρια οχήματα

Αθανάσιος (Νάσος) Γρηγορόπουλος

Επιβλέπων:
Καθ. Σπυρίδων-Γεράσιμος Λάλης

Διατριβή η οποία υποβλήθηκε για τη μερική εκπλήρωση των υποχρεώσεων απόκτησης του Διδακτορικού Διπλώματος

στο

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Επταμελής εξεταστική επιτροπή:
Σπυρίδων-Γεράσιμος Λάλης, Καθηγητής, Πανεπιστήμιο Θεσσαλίας
Χρήστος Αντωνόπουλος, Αναπληρωτής Καθηγητής, Πανεπιστήμιο Θεσσαλίας
Αντώνιος Αργυρίου, Αναπληρωτής Καθηγητής, Πανεπιστήμιο Θεσσαλίας
Νικόλαος Μπέλλας, Καθηγητής, Πανεπιστήμιο Θεσσαλίας
Δημήτριος Κατσαρός, Αναπληρωτής Καθηγητής, Πανεπιστήμιο Θεσσαλίας
Ευστάθιος Χατζηευθυμιάδης, Καθηγητής, Πανεπιστήμιο Αθηνών
Κωνσταντίνος Μαγκούτης, Αναπληρωτής Καθηγητής, Πανεπιστήμιο Κρήτης

12 Απριλίου 2022

Declaration of Authorship

Being fully aware of the implications of copyright laws, I expressly state that this PH.D. dissertation, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

Signed:

Date: 12/4/2022

"The cost of a thing is the amount of what I will call life which is required to be exchanged for it, immediately or in the long run."

Henry David Thoreau

UNIVERSITY OF THESSALY

Abstract

Department of Electrical and Computer Engineering

Doctor of Philosophy

System Support for the Fault Tolerance, Testing and Orchestration of Drone Applications

by Athanasios (Nasos) GRIGOROPOULOS

The continuous technological advances in embedded platforms and control systems have made unmanned aerial vehicles (drones) on the one hand more autonomous in their operation, combining computing capabilities, advanced navigation systems and obstacle avoidance systems, and on the other hand more affordable, paving the way for their use in many application fields, such as surveillance, monitoring and cargo delivery. Although drones can now be programmatically controlled via high-level commands, their smooth integration in the existing cyber-physical infrastructure is hampered by several factors including inherent safety and privacy concerns and the lack of automation in the (complete) drone operation lifecycle. To address this issue, in this thesis we present system-level mechanisms for strengthening the robustness of drone applications and promoting a more structured, managed and automated application development and deployment approach, properly integrated with the cloud and edge computing paradigms.

In the first part of the thesis, we focus on tolerating failures critical to the target mission in application scenarios where a centralized mission controller entity is responsible for executing the application logic by coordinating a team of drones via high-level commands. We propose an active replication scheme combined with checkpointing and logging of communication and system status information for tolerating fail-stop failures, where the controller simply stops its operation. We show that in case of deterministic application programs the overhead introduced by our solution occurs only in case of drone failures and depends on the size of the logs that need to be exchanged between the controller replicas. Our solution also supports non-deterministic applications through a semi-active replication approach, in which case the overhead is analogous to the size of the execution state produced during the non-deterministic operations. Next, we use active replication with an agreement protocol to address Byzantine failures, the most general type of failures, where the controller behaves arbitrarily due to malicious attacks or hardware and software errors. Our approach relies on synchronous communication with signed messages and requires $N = 2 \times f + 1$ replicas to tolerate up to f failures. We show that the introduced overhead comes mainly from the agreement protocol and can be greatly reduced by using a faster network for the communication between the replicas of the coordinator (a realistic assumption). Also, while the cost for typical setups is relatively high, at the range of tens of milliseconds, it is not prohibitive since the mission controller is not involved in tight control loops of the drones themselves.

In the second part of the thesis, we look at how to assist drone application developers and users in the application deployment, monitoring and testing processes while ensuring that certain safety and privacy restrictions are not violated. To address these needs in a cost-effective manner, we adopt the platform as a service (PaaS) paradigm of cloud computing and view drones as a resource that can be accessed through a shared infrastructure. Firstly, we present a holistic approach towards supporting a more reliable managed operation of single-component drone applications through a software platform that takes care of the automated deployment and controlled execution, coupled with corresponding simulation and digital twin support. Differently to other efforts, the monitoring mechanisms do not concern only the "where" drones are allowed to operate, but also take account of the "how" their onboard sensing and actuation equipment is used. We also demonstrate the provided functionality through representative case studies using both the simulation environment where we perform a variety of tests before deployment, and the digital twin setup where a virtual representation runs in parallel with the application in the real world, providing the ability to evaluate, verify and even predict the behavior of the system. In the following, we expand various dimensions of the concept to distributed, component-based applications that span over the entire system continuum, including drones, edge nodes and the cloud. We present an orchestration framework that enables the high-level description of application requirements through suitable specifications that accompany the component code, supports the drone mission-aware deployment of the components, promotes the efficient communication between interacting components by transparently exploiting wireless networking technologies for direct communication and ensures safety- and privacy-preserving operations through the policy-based access to mobility and sensor services. We demonstrate the provided functionality via both field tests and the simulation environment. Further, through extensive performance evaluation, we show that our implementation has an acceptable resource footprint making it suitable even for the constrained computing platforms found on drones and edge nodes.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Περίληψη

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διδακτορικό Δίπλωμα

Υποστήριξη σε επίπεδο συστήματος για την ανοχή βλαβών, δοκιμαστική λειτουργία και ενορχήστρωση σε εφαρμογές με μη επανδρωμένα εναέρια οχήματα

Αθανάσιος (Νάσος) Γρηγορόπουλος

Οι συνεχείς τεχνολογικές εξελίξεις σε ενσωματωμένες πλατφόρμες και συστήματα ελέγχου έχουν καταστήσει τα μη επανδρωμένα εναέρια οχήματα (drones) αφενός πιο αυτόνομα στη λειτουργία τους, συνδυάζοντας υπολογιστικές δυνατότητες, προηγμένα συστήματα πλοήγησης και συστήματα αποφυγής εμποδίων, και αφετέρου πιο προσιτά από οικονομικής άποψης, διευρύνοντας το πεδίο χρήσης τους σε διάφορους τομείς εφαρμογών, όπως η επιτήρηση, η παρακολούθηση και η μεταφορά-παράδοση φορτίων/αγαθών. Αν και τα drones μπορούν πλέον να ελέγχονται προγραμματιστικά μέσω εντολών υψηλού επιπέδου, η ομαλή ενσωμάτωσή τους στην υπάρχουσα κυβερνο-φυσική υποδομή παρεμποδίζεται από διάφορους παράγοντες, όπως είναι οι εγγενείς ανησυχίες σχετικά με τους κινδύνους ασφαλείας και τη προστασία της ιδιωτικότητας καθώς και η έλλειψη αυτοματισμού στον (πλήρη) κύκλο λειτουργίας του drone. Για την αντιμετώπιση των ανωτέρω ζητημάτων, σε αυτή τη διατριβή παρουσιάζουμε μηχανισμούς σε επίπεδο συστήματος για την ενίσχυση της αξιοπιστίας/ευρωστίας των εφαρμογών drone και την προώθηση μιας πιο δομημένης, ελεγχόμενης και αυτοματοποιημένης προσέγγισης ανάπτυξης και εγκατάστασης εφαρμογών, η οποία θα είναι αρμονικά συνδεδεμένη με τα υπολογιστικά πρότυπα που ακολουθούνται στο υπολογιστικό νέφος (cloud) και στα άκρα του δικτύου (edge).

Στο πρώτο μέρος της διατριβής, εστιάζουμε στην ανοχή σφαλμάτων κρίσιμων για την πρόοδο της αποστολής σε σενάρια εφαρμογών όπου μια κεντρική υπολογιστική οντότητα ελεγκτή αποστολής είναι υπεύθυνη για την εκτέλεση της λογικής της εφαρμογής συντονίζοντας μια ομάδα drones μέσω εντολών υψηλού επιπέδου. Προτείνουμε ένα σχήμα ενεργού πλεονασμού (active replication) σε συνδυασμό με τεχνικές αποθήκευσης σημείων ελέγχου της κατάστασης του συστήματος (checkpointing) και καταγραφής πληροφοριών επικοινωνίας (logging), με σκοπό την ανοχή βλαβών αποτυχίας-σταματήματος (fail-stop failures), όπου ο ελεγκτής απλώς σταματά τη λειτουργία του. Δείχνουμε ότι σε περίπτωση ντετερμινιστικών προγραμμάτων εφαρμογής, η λύση μας εισάγει επιβάρυνση μόνο σε περίπτωση που παρουσιάσει βλάβη κάποιο drone και εξαρτάται από το μέγεθος των αρχείων καταγραφής που πρέπει να ανταλλαχθούν μεταξύ των αντιγράφων του ελεγκτή. Η λύση μας υποστηρίζει επίσης μη ντετερμινιστικές εφαρμογές μέσω μιας προσέγγισης ημι-ενεργού πλεονασμού, όπου το κόστος είναι ανάλογο με το μέγεθος της κατάστασης εκτέλεσης που παράγεται κατά τις μη ντετερμινιστικές λειτουργίες. Στη συνέχεια, χρησιμοποιούμε την ενεργή αναπαραγωγή με ένα πρωτόκολλο συμφωνίας για την αντιμετώπιση Βυζαντινών βλαβών (Byzantine failures), του πιο γενικού τύπου αστοχιών, όπου ο ελεγκτής μπορεί να συμπεριφέρεται αυθαίρετα λόγω κακόβουλων επιθέσεων ή σφαλμάτων υλικού και λογισμικού. Η προσέγγισή μας βασίζεται σε υποθέσεις σύγχρονης επικοινωνία με υπογεγραμμένα μηνύματα και απαιτεί $N = 2 \times f + 1$ αντίγραφα για την ανοχή έως

και f βλαβών. Δείχνουμε ότι το εισαγόμενο συνολικό κόστος προέρχεται κυρίως από το πρωτόκολλο συμφωνίας και μπορεί να μειωθεί σημαντικά χρησιμοποιώντας ένα ταχύτερο δίκτυο για την επικοινωνία μεταξύ των αντιγράφων του συντονιστή - κάτι το οποίο αποτελεί μια ρεαλιστική υπόθεση. Επίσης, ενώ το κόστος για τυπικές συνθήκες/ρυθμίσεις είναι σχετικά υψηλό, στο εύρος των δεκάδων χιλιοστών του δευτερολέπτου, δεν είναι απαγορευτικό δεδομένου ότι ο ελεγκτής αποστολής δεν εμπλέκεται σε χαμηλού επιπέδου βρόχους ελέγχου σχετιζόμενους με την πτητική λειτουργία του drone.

Στο δεύτερο μέρος της διατριβής, εξετάζουμε τρόπους και τεχνικές που θα βοηθήσουν τους προγραμματιστές και τους χρήστες εφαρμογών για drone στις διαδικασίες ανάπτυξης, παρακολούθησης και δοκιμής των εφαρμογών, διασφαλίζοντας παράλληλα ότι δεν παραβιάζονται συγκεκριμένοι περιορισμοί ασφάλειας και ιδιωτικότητας. Για να αντιμετωπίσουμε αυτές τις ανάγκες με έναν οικονομικά αποδοτικό/βιώσιμο τρόπο, υιοθετούμε την προσέγγιση της πλατφόρμας ως υπηρεσία (PaaS) που χρησιμοποιείται στο cloud και αντιμετωπίζουμε τα drones ως άλλον έναν υπολογιστικό πόρο στον οποίο μπορεί κάποιος να έχει πρόσβαση μέσω μιας κοινόχρηστης υποδομής. Αρχικά, παρουσιάζουμε μια ολιστική προσέγγιση για την υποστήριξη της πιο αξιόπιστης και ελεγχόμενης λειτουργίας «μονοσυστατικών» εφαρμογών για drones μέσω μιας πλατφόρμας λογισμικού που φροντίζει για την αυτοματοποιημένη εγκατάσταση και την ελεγχόμενη εκτέλεσή τους, σε συνδυασμό με την παροχή κατάλληλης υποστήριξης προσομοίωσης και τεχνολογίας ψηφιακών διδύμων. Σε αντίθεση με άλλες προσπάθειες, οι μηχανισμοί παρακολούθησης δεν αφορούν μόνο το «πού» επιτρέπεται να λειτουργούν τα drones, αλλά λαμβάνουν επιπλέον υπόψη το «πώς» χρησιμοποιείται ο εξοπλισμός αισθητήρων και ενεργοποιητών που διαθέτουν. Επιδεικνύουμε την παρεχόμενη λειτουργικότητα μέσω αντιπροσωπευτικών παραδειγμάτων χρήσης χρησιμοποιώντας τόσο το περιβάλλον προσομοίωσης όπου εκτελούμε ένα εύρος δοκιμών πριν από την εγκατάσταση της εφαρμογής, όσο και των ψηφιακών διδύμων όπου μια εικονική αναπαράσταση του drone εκτελείται παράλληλα με την εφαρμογή στον πραγματικό κόσμο, παρέχοντας τη δυνατότητα αξιολόγησης, επαλήθευσης αλλά και πρόβλεψης της συμπεριφοράς του συστήματος. Στη συνέχεια, επεκτείνουμε διάφορες διαστάσεις αυτής της γενικής ιδέας σε κατανεμημένες εφαρμογές που αποτελούνται από πολλαπλά τμήματα τα οποία μπορούν να εγκατασταθούν σε ολόκληρο το συνεχές των υπολογιστικών συστημάτων, συμπεριλαμβανομένων drones, κόμβων στο edge και του cloud. Παρουσιάζουμε ένα πλαίσιο ενορχήστρωσης που επιτρέπει την περιγραφή υψηλού επιπέδου των απαιτήσεων της εφαρμογής μέσω κατάλληλων προδιαγραφών που συνοδεύουν τον κώδικα των τμημάτων της εφαρμογής, υποστηρίζει την εγκατάσταση των τμημάτων της εφαρμογής λαμβάνοντας υπόψη την αποστολή του drone, προάγει την αποτελεσματική επικοινωνία μεταξύ αλληλεπιδρώντων τμημάτων αξιοποιώντας τεχνολογίες ασύρματης δικτύωσης για άμεση επικοινωνία με διαφανή τρόπο και διασφαλίζει τη διατήρηση της ασφάλειας και την προστασία της ιδιωτικότητας μέσω της πρόσβασης των εφαρμογών σε υπηρεσίες κινητικότητας και αισθητήρων βάσει πολιτικών. Επιδεικνύουμε την παρεχόμενη λειτουργικότητα τόσο μέσω δοκιμών στο πεδίο όσο και μέσω του περιβάλλοντος προσομοίωσης. Επιπλέον, μέσω εκτεταμένης αξιολόγησης απόδοσης, δείχνουμε ότι η υλοποίησή μας έχει αποδεκτές υπολογιστικές απαιτήσεις, καθιστώντας την κατάλληλη ακόμη και για τις περιορισμένων δυνατοτήτων υπολογιστικές πλατφόρμες που συναντώνται σε drones και σε κόμβους στο edge.

Acknowledgements

I first and foremost thank my advisor Spyros Lalis. He has been exceptional in mentoring me over the years, from my undergraduate studies to the completion of this thesis, and I owe to him most of what I know about doing research. I am also grateful to Assoc. Prof. Christos D. Antonopoulos, Assoc. Prof. Antonios Argyriou, Prof. Nikolaos Bellas, Assoc. Prof. Dimitrios Katsaros, Prof. Stathes Hadjiefthymiades and Assoc. Prof. Kostas Magoutis for accepting to participate in the Examination Committee of my thesis and approving my work.

Many thanks go to my long-time colleague Manos Koutsoubelias for being a great discussion partner on a wide range of subjects; sharing my ideas and doing research with him for the past decade has been a wonderful experience. Special thanks also go to my friend and colleague Konstantinos Parasyris for the enjoyable moments that helped ease the burden of research. My thanks are extended to all members of the Computer Systems Lab for contributing to a pleasant working environment.

On a more personal level, I wish to thank my closest friends and especially Giannis K., Dimitris D., Lazaros L. and Evangelia N. for being supportive in all of my decisions for over fifteen years and for making life outside of studies and research really interesting.

Finally, my deepest gratitude goes to my family: my parents, Spyros and Evanthia, and my sister Alexia-Iris. Their unconditional love, moral encouragement and belief in me throughout my life and my long academic pursuits have been invaluable. The least I can say to them is a big “thank you”. Last, I am extremely grateful to Eva Evangeliou, who I already consider part of my family. Eva has supported me throughout the years, listened my research presentations at home and helped me improve them, but most importantly has helped me to evolve personally.

The work in this thesis has been supported by various funding agencies to which I am grateful for offering the financial means to stay focused to my research and pursue my PhD uninterrupted. These include the European Commission through the RAWFIE project (Road-, Air- and Water- based Future Internet Experimentation) of the Horizon 2020 Framework Programme (Grant Agreement No 645220, period: 2016-2018), the Greek Secretariat for Research and Development through the PV-Auto-Scout project under the Research–Create–Innovate call (code T1EDK-02435, period: 2018-2020 & 2021-2022) and the University of Thessaly through the Research, Innovation and Excellence (DEKA) Scholarship Program (period: 2020-2021).

Publications

The results, the ideas and figures are included in the following publications:

[1] Nasos Grigoropoulos and Spyros Lalis, "Fractus: Orchestration of Distributed Applications in the Drone-Edge-Cloud Continuum", in *Proc. 46th IEEE Annual Computers, Software, and Applications Conference (COMPSAC 2022)*, to appear, June 27–July 1, 2022.

[2] Nasos Grigoropoulos and Spyros Lalis, "Simulation and Digital Twin Support for Managed Drone Applications", in *Proc. 24th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2020)*, Prague, Czech Republic, pp. 198-205, September 14-16, 2020.

[3] Nasos Grigoropoulos and Spyros Lalis, "Flexible Deployment and Enforcement of Flight and Privacy Restrictions for Drone Applications", in *Proc. 50th IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 2020), International Workshop on Safety and Security of Intelligent Vehicles (SSIV)*, Valencia, Spain, pp. 110-117, June 29 - July 2, 2020.

[4] Nasos Grigoropoulos, Manos Koutsoubelias and Spyros Lalis, "Byzantine Fault Tolerance for Centrally Coordinated Missions with Unmanned Vehicles", in *Proc. 17th ACM International Conference on Computing Frontiers (CF 2020)*, Catania, Sicily, Italy, pp. 165-173, May 11-13, 2020.

[5] Nasos Grigoropoulos, Manos Koutsoubelias and Spyros Lalis, "Active Replication for Centrally Coordinated Teams of Autonomous Vehicles", in *Proc. 15th International Conference on Distributed Computing in Sensor Systems (DCOSS 2019)*, Santorini, Greece, pp. 114–122, May 29-31, 2019.

In addition, our research efforts within the same period led to the following publications that are relevant to the research in this thesis but not directly related to the main subject:

[6] Manos Koutsoubelias, Nasos Grigoropoulos, Giorgos Polychronis, Giannis Badakis and Spyros Lalis, "System Architecture for Autonomous Drone-based Remote Sensing", in *Proc. 18th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2021)*, Virtual, pp. 220–242, November 8-11, 2021.

[7] Manos Koutsoubelias, Nasos Grigoropoulos and Spyros Lalis, "A Modular Simulation Environment for Multiple UAVs with Virtual WiFi and Sensing Capability", in *Proc. 2018 IEEE Sensors Applications Symposium (SAS 2018)*, Seoul, South Korea, March 12-14, 2018.

[8] Manos Koutsoubelias, Nasos Grigoropoulos and Spyros Lalis, "Virtual Sensor Services for Simulated Mobile Nodes", in *Proc. 2017 IEEE Sensors Applications Symposium (SAS 2017)*, Glassboro, NJ, USA, March 13-15, 2017.

Contents

List of Figures

List of Tables

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
BFT	**B**yzantine **F**ault **T**olerance
BLE	**B**luetooth **L**ow **E**nergy
CPS	**C**yber-**P**hysical **S**ystems
CPU	**C**entral **P**rocessing **U**nit
CRUD	**C**reate, **R**ead, **U**pdate and **D**elete
CSMA	**C**arrier-**S**ense **M**ultiple **A**ccess
DaaS	**D**rone **as a** **S**ervice
DMTCP	**D**istributed **M**ulti**T**hreaded **C**heck**P**ointing
DT	**D**igital **T**win
FaaS	**F**unction **as a** **S**ervice
FDM	**F**light **D**ynamics **M**odel
FIFO	**F**ist **in**, **F**irst **o**ut
GCBRR	**G**roup **C**oordinated **B**roadcast-Based **R**equest-**R**eply
GCS	**G**round **C**ontrol **S**tation
HBICT	**H**ash **B**ased **I**ncremental **C**heckpointing **T**ool
HITL	**H**ardware-**in**-**t**he-**L**oop
IMU	**I**nertial **M**easurement **U**nit
IoT	**I**nternet **o**f **T**hings
IP	**I**nternet **P**rotocol
JSON	**J**ava**S**cript **O**bject **N**otation
KVM	**K**ernel-Based **V**irtual **M**achine
LAN	**L**ocal **A**rea **N**etwork
LTE	**L**ong **T**erm **E**volution
LXC	**L**inu**x** **C**ontainer
LXD	**L**inu**x** Container **D**aemon
MAVLink	**M**icro **A**ir **V**ehicle **Link**
MAVProxy	**MAV**Link **Proxy**
NFV	**N**etwork **F**unctions **V**irtualization
PaaS	**P**latform **as a** **S**ervice
PBFT	**P**ractical **B**yzantine **F**ault **T**olerance
PHC	**P**TP **H**ardware **C**lock
PTP	**P**recision **T**ime **P**rotocol
QoS	**Q**uality **o**f **S**ervice
REST	**Re**presentational **S**tate **T**ransfer
ROS	**R**obot **O**perating **S**ystem
RPC	**R**emote **P**rocedure **C**all
SDN	**S**oftware **D**efined **N**etworking
SITL	**S**oftware-**in**-**t**he-**L**oop
SSL	**S**ecure **S**ockets **L**ayer
TCP	**T**ransmission **C**ontrol **P**rotocol
TeCoLa	**Te**am **Co**ordination **La**nguage

TLS	**T**ransport **L**ayer **S**ecurity
UAV	**U**nmanned **A**erial **V**ehicles
UDP	**U**nreliable **D**atagram **P**rotocol
UE	**U**ser **E**quipment
USB	**U**niversal **S**erial **B**us
UV	**U**nmanned **V**ehicles
VIP	**V**irtual **IP**
VM	**V**irtual **M**achine
VPN	**V**irtual **P**rivate **N**etwork
v-GS	**v**irtual **G**round **S**tation
WiFi	**Wi**reless **Fi**delity
WLAN	**W**ireless **LAN**
YAML	**Y**AML **A**in't **M**arkup **L**anguage

Chapter 1

Introduction

1.1 Motivation and Problem Statement

The continuous advances in sensors, control systems and embedded computing have enabled the development of many different types of unmanned vehicles (UVs) with considerable navigation and obstacle avoidance capabilities, leading to higher levels of autonomy [172]. Such platforms constitute a radical expansion of the cyber-physical systems landscape, which revolutionizes existing applications and will possibly give rise to new ones. In particular, unmanned aerial vehicles (UAVs), commonly known as drones, while in the past were used almost exclusively either for military and law enforcement purposes or by hobbyists, are now increasingly being employed in various application domains, spanning from surveillance, agriculture and cargo transport to the inspection of critical infrastructure and smart-city applications [157, 11]. Moreover, drones have become quite significant in terms of business. The drone market is estimated at USD 27.4 billion in 2021 and is projected to reach USD 58.4 billion by 2026, at an annual growth rate of 16.4%, with the key factors driving the market growth being the increasing demand for drones in civil and commercial sectors, as well as the rising demand for contactless last mile deliveries of medical supplies and other essentials [111]. In this thesis, we focus on smaller drones that can perform flexible maneuvers, such as typical multicopters with vertical take-off and landing capability. However, there are various challenges than need to be addressed to realize the full potential of drone technology and overcome safety and privacy concerns.

Automation in the full drone operation cycle. Until recently, the majority of drone missions used to be human-operated and required substantial control with specialized ground control stations. The rise of drone platforms that can be controlled in a more automatic way through high-level commands issued by computer programs, led to the creation of several programming abstractions and languages in order to simplify the development of drone missions/operations. These programming frameworks span from the control of a single drone [149, 38], to the control of multiple drones, either in a distributed way [130], or in a centrally coordinated manner [85]. However, drone applications to a significant extent still require human intervention in several stages of their operation, ranging from the application deployment to the toleration of non-catastrophic failures, hindering the creation of fully automated drone-based systems.

Cost of drone ownership and operation. While commercial off-the-shelf drones have become relatively cheap, specialized drones built with durable materials or carrying high-end equipment remain financially unapproachable for most small-to-medium companies [68]. On top of the acquisition cost, come the licensing, insurance and maintenance costs, which are non-negligible [40]. Moreover, as any technology-based product, drone platforms may quickly become obsolete and frequent upgrades will most likely be needed if one wishes to have cutting-edge equipment. Thus, all the above costs have to be amortized within a relatively short period of time, which is hard unless one achieves a high utilization of the drone.

Management of drone operations. Recent regulations from aviation safety agencies, e.g., European Union's EASA [39], set the basic framework for the safe operation of drones, paving the way for their integration even in the urban environment, which was long ago believed to be science fiction. However, the upcoming coexistence of multiple drones flying over citizens and private properties raises several safety and privacy issues due to the possible crashes, collisions and the uncontrolled usage of the drone's onboard sensing equipment [170, 177, 118]. Even though most countries have formal processes for submitting flight plans and getting approval, in many cases there are no well-integrated mechanisms ensuring that the approved flight plan is followed. Moreover, low altitude airspace management systems [121, 158] focus almost exclusively to safety-related spatial restrictions and do not consider privacy-specific constraints related to the sensors usage. Inevitably, this leads to skepticism regarding drone-based systems, limited public acceptance and even to extreme reactions by people opposed to drone usage [120].

Integration with existing computing infrastructure. Even though drones can carry powerful, small form factor onboard computers, in various cases it is essential to interact with existing ICT infrastructure in order to augment their computing capabilities by offloading heavyweight computations and utilizing the ample persistent storage, as well as with nearby Internet of Things (IoT) edge deployments to incorporate data from different sources and achieve better responsiveness [19]. Ideally, this cooperation should be smooth, straightforward and should take place under a variety of wireless communication technologies transparently to the application programmer. In practice, however, such kind of applications are currently being deployed in an ad-hoc manner requiring high human involvement and are by nature customized for specific setups, which makes them inflexible in adapting to different environments or changing conditions.

1.2 Contributions

The aim of this thesis is to address the challenges mentioned above by answering the following questions:

1. How to offer more dependability to drone operations and strengthen their overall automation capability?
2. How to support the flexible and automated deployment and testing of drone-based applications while ensuring their smooth integration in the cyber-physical infrastructure?

Towards these goals, we have designed, developed and evaluated system-level mechanisms focused in two distinct directions: (i) the toleration of failures that are critical to the application progress, in order to minimize the need for human intervention and thus make automated drone missions more robust; and (ii) the promotion of a structured and managed application deployment, operation and testing approach that enforces the compliance to safety and privacy restrictions and enables the efficient and transparent interaction with other computing resources that are available in the cloud as well as at the edge. In the following we give an overview of the main contributions of this thesis in these two directions.

1.2.1 Fault tolerance techniques in coordinated drone missions

The deployment of multiple drones in order to perform a mission as a team can offer increased benefits in several application domains, e.g., reduction of completion time or improved quality of the end result. While a certain degree of self-organization can be achieved using swarming techniques, these typically assume a large number of homogeneous drones with relatively limited sensing and processing capabilities and rely on quite extensive communication between the individual drones, which are assumed to be in proximity with each other [151]. However, several real-world applications can be efficiently supported by employing small teams consisting of a few but more powerful drones that can navigate autonomously and may have different sensing/actuation capabilities. In our work, we consider the case where such drones are coordinated by a distinguished computing entity, the mission controller, which runs the mission program, collects information from all drones and issues high-level control commands to them, through structured remote service interfaces.

This centralized approach offers many advantages in terms of programmability and management of the available sensing/actuation resources. Also, individual drone failures are not critical to the application progress as the mission controller can redistribute the tasks of the failed drone to the remaining ones. However, in this system model, the mission controller constitutes the single most important component for the continuity of the mission. Thus, to make the system more dependable and avoid manual/human intervention, it becomes crucial to tolerate failures of the mission controller in a transparent way.

At first, we focus on tolerating fail-stop failures, where the controller simply stops its operation. We define the properties that should be satisfied to ensure consistency and highlight the issues that need to be addressed. For deterministic mission programs, we propose an active replication scheme, where the program is actively executed by multiple instances of the mission controller, combined with logging of both communication and system status information. While the fundamental principles of active replication have been laid out a long time ago, this problem has been traditionally studied for the server/service side of computer systems [155]. In our case however, the mission controller invokes the drones acting as their client, mandating a different approach. We show that the introduced overhead occurs only in case of drone failures and depends on the size of the logs that need to be exchanged between the mission controller replicas. We also expand the proposed solution to non-deterministic applications through a semi-active replication approach, where non-deterministic operations are executed by a distinguished mission controller replica which checkpoints the locally produced execution state and transfers it to the other replicas. In this case, the overhead is analogous to the size of the checkpoints.

Next, we address Byzantine failures, the most general type of failures, where the mission controller may behave arbitrarily due to malicious attacks or hardware and software errors. We specify the required properties and propose an active replication approach with an agreement protocol adapted to characteristics of this particular type of system. Our approach relies on synchronous communication with signed messages and requires $N = 2 \times f + 1$ replicas to tolerate f failures. We informally argue about the correctness of the proposed solution and show that the introduced overhead comes mainly from the agreement protocol, which can be greatly reduced by using a faster network for the communication between the replicas of the controller (which is a realistic assumption). Also, while the cost for typical setups is relatively high, at the range of tens of milliseconds, it is not prohibitive since the mission controller is not involved in any tight control loops of the drones (these run locally on each drone).

1.2.2 Managed operation, testing and integration of drone applications

To effectively operationalize drones in the context of next-generation applications, it is important to shift from the currently established ad-hoc usage model to a more structured and managed application development and deployment approach, properly integrated with the edge- and cloud-computing paradigms. To achieve this, we envision drones as a proper infrastructure which can be used to support different applications in a flexible and efficient way through a platform as a service (PaaS) system that relieves the application developer/user from all the resource allocation, deployment, connectivity and monitoring issues.

However, drones are special: they are mobile, rely on wireless communication, feature different sensors and resource-constrained embedded computing platforms, and have limited operational autonomy (especially the widely used multicopters). For these reasons, drones cannot be just plugged in a cloud or edge system as "simple" nodes. They must be handled in a special way, taking into account the strong geographic dimension of the corresponding applications, while dealing with the safety and privacy issues and providing suitable testing and monitoring mechanisms.

At first, we present a holistic approach towards supporting a more reliable managed operation of single-component drone applications through a software platform that takes care of their automated deployment and controlled execution, coupled with corresponding simulation and digital twin support for detecting bugs before deployment and indicating possible malfunctions during operation in the real world, respectively. We provide a high-level description of the concept and discuss the most important aspects of a proof-of-concept implementation. Differently to other efforts, the monitoring mechanisms do not focus only spatial-related restrictions where drones are allowed to operate, but also concern the usage of their onboard sensing and actuation equipment. We also demonstrate the provided functionality through representative case studies using both the simulation environment where we perform a variety of tests before deployment, and the digital twin setup where a virtual representation of the drone runs in parallel with the application in the real world, providing the ability to evaluate, verify and even predict the behavior of the system.

In the following, we expand the concept to distributed, component-based applications that span over the entire system landscape, including drones, static edge nodes and the cloud. We present an orchestration framework that enables the high-level description of application requirements through suitable specifications that accompany the component code and takes care of the automatic deployment and interconnection of the application components across the drone-edge-cloud continuum. Internally, resource allocation and component deployment are mission-aware, based on the geographic area where the application will use the drone, while connectivity between application components is managed transparently, exploiting ad-hoc local networking opportunities between drone and edge nodes hosting interacting components. In addition, critical drone and edge functions, such as mobility and sensor operations, are accessed by the application in a controlled way, through interfaces that offer secure, safety- and privacy-preserving operations, driven by structured policies. We demonstrate the provided functionality via both field tests and the simulation environment and through extensive performance evaluation we show that our implementation has an acceptable resource footprint making it suitable even for the constrained computing platforms found on drones and edge nodes.

1.3 Thesis Outline

The rest of the thesis is structured as follows.

In Chapter 2, we present the experimental tools, consisting of a complete simulation environment and a drone testbed, that are used throughout this thesis for the evaluation of the various mechanisms and algorithms we propose.

Chapters 3 through 6 constitute the first part of the thesis and deal with the fault tolerance techniques in multi-drone application scenarios. Chapter 3 presents the system model for coordinated drone missions that we assume in our work and the extensions needed to support the active replication of the mission program. Chapter 4 discusses the active replication approach for tolerating fail-stop failures in both deterministic and non-deterministic mission programs. Chapter 5 describes the replication scheme with the assumptions and mechanisms for achieving Byzantine fault tolerance. Chapter 6 gives an overview of related work regarding the toleration of both fail-stop and Byzantine failures as well as indicative fault tolerance mechanisms that are specific to robotic systems.

Chapters 7 through 9 comprise the second part of the thesis and focus on the managed operation, testing and integration of drone applications. Chapter 7 discusses our holistic approach for the managed operation of drone applications which consists of the software platform that supports their automated deployment and controlled execution, and the simulation and digital twin support. Chapter 8 presents the framework for the orchestration of distributed applications in the drone-edge-cloud continuum. Chapter 9 provides an overview of related work in the areas of edge and drone-specific application deployment and testing and identifies our main differentiation points.

Finally, Chapter 10 concludes the thesis summarizing our key contributions and pointing out directions for future work.

Chapter 2

Experimental Tools

2.1 Introduction and Outline

In this chapter, we describe the main tools we have developed and use for the prototype development and the experimental evaluation of the mechanisms we introduce. Testing is an important part of the software development cycle. Especially, in cyber-physical systems that involve autonomous vehicles like drones, which can directly interact with and affect the environment, testing becomes an integral part of the development process and has to be carried out systematically. To this end, we use both a hardware-based testbed and an integrated simulation environment.

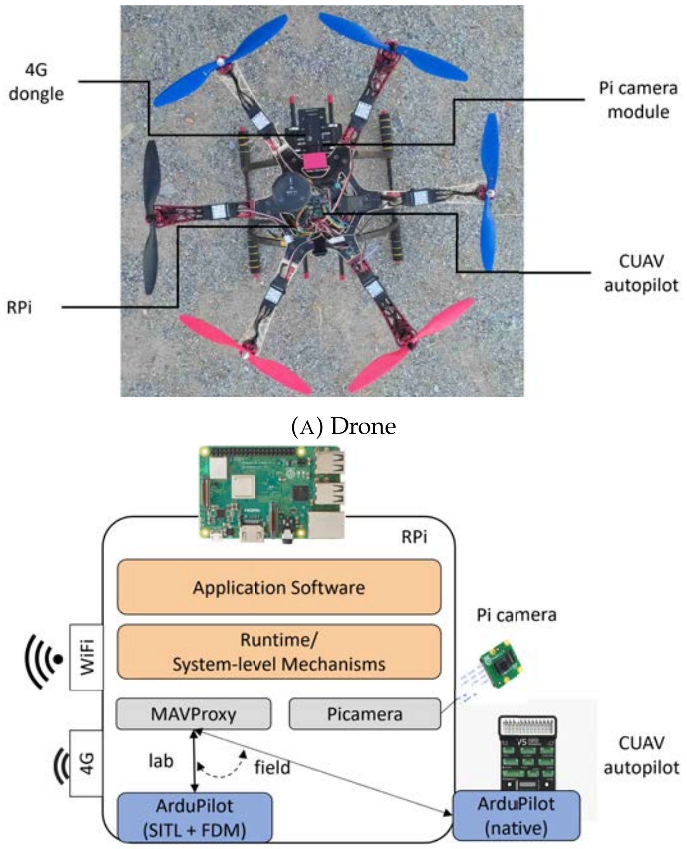
The hardware-based testbed consists of typical platforms of drones, embedded edge nodes and controller computers. In the lab setup, it can be used for measuring the various overheads in terms of computing resources required by the actual platforms that would be used in a real experiment. In the field setup, it can be used for conducting real-world experiments to gain valuable insight regarding the overall system operation and functionality achieved under real conditions.

However, real-world experimentation with drones usually comes with significant overhead and can be quite costly in terms of person hours. Also, in case of failures, drones are subject to severe damage that may require costly repairs. The issue of safety makes things even harder because tests have to get formal approval from civil authorities and extra care must be taken to ensure that drones will not run out of control due to software bugs or a hardware malfunction. Considering the need for extensive evaluation regarding both the applications and the system software, we have developed a simulation environment that can assist during the early stages of development by allowing the fast, controlled, secure, cheap and flexible experimentation of complex scenarios and system configurations with multiple communicating drones and/or edge nodes.

The rest of the chapter is organized as follows. Section 2.2 presents the basic components of the hardware testbed setup. Section 2.3 briefly discusses the capabilities and the design of the simulation setup. Finally, Section 2.4 concludes the chapter.

2.2 Hardware Testbed

Controller. All software related to the mission control and the drone management is hosted in a typical server machine that effectively acts as a ground control station (GCS) for the mission at hand. In most experiments we use a Dell Precision



(A) Drone

(B) Software/hardware organization

FIGURE 2.1: Drone setup used in the field and lab experiments

Tower 5810 server with Intel Xeon CPU E5-1620 v4 at 3.50GHz and 16GB RAM [35], running a standard Linux distribution (Ubuntu 18.04).

Drone. The drone used in our experiments, shown in Figure 2.1a, is a custom-made hexacopter with a CUAV V5 nano autopilot board [168] running the popular ArduPilot autopilot software for multicopters (Copter v4.0) [7]. The drone has as a separate companion board a Raspberry Pi 3 Model B [145] with a quad-core ARM Cortex A53 (ARMv8) processor at 1.2 GHz and 1GB of RAM, running the Debian-based Raspberry Pi OS Buster, which is the officially supported Linux distribution for the RPi. The RPi is connected to the autopilot board over serial and runs all the system-level software we introduce as well as any application-level software, as shown in Figure 2.1b. For controlling the mobility of the drone, we employ a user-level library communicating with the autopilot using MAVLink [112] messages, which, depending on the scenario, can be one of the low-level C and Python (pymavlink) MAVLink libraries [136, 137], or a higher-level API like DroneKit [38], MAVSDK [115] and ROS [149] through the MAVROS communication node [114]. The drone also features an 8MP Raspberry Pi Camera Module 2 [144], which is accessed using the picamera library [129]. From a communication viewpoint, apart from the RPi's WiFi interface that can be configured for wireless LAN communication in infrastructure or ad-hoc mode, the drone is also equipped with a 4G/LTE USB modem [63] for interacting over Internet.

Edge nodes. In experiments where edge nodes are required, we employ additional, standalone RPis that are equipped with the same camera module and have a configuration similar with the one described for the drone's companion computer, except from the autopilot-related parts. In addition, depending on the testing scenario, these nodes can also use the Ethernet interface for faster and more stable communication/data transfers with other machines that play the role of additional edge/cloud infrastructure.

Field setup. This setup is used to perform real-world experiments to test and verify the functionality of our mechanisms in flight conditions. Typically, the drone is configured to use the 4G/LTE modem in order to interact with the controller computer over the Internet via a VPN connection. In cases where edge nodes are deployed, these can connect to the VPN through their Ethernet interface. In addition, the edge nodes can interact with the drone directly over WiFi.

Lab setup. This setup is mainly used to conduct a wide range of tests and performance measurements in the lab, without having to fly the drone. To this end, on the drone we disconnect the RPi from the autopilot board and configure MAVProxy to use the official software-in-the-loop (SITL) configuration of ArduPilot [8] with its integrated flight dynamics model, which we also run on the drone's RPi, as shown in Figure 2.1b. Apart from this, the drone setup is identical to the one used in the field.

2.3 Simulation Environment

The simulation environment is based on the AeroLoop system [81]. AeroLoop was conceived and developed by the Computer Systems Lab [30] of the Department of Electrical and Computer Engineering of University of Thessaly, in the AeroLoop project [1] that was funded through the open calls of the H2020 FIRE+ RAWFIE project [146]. The AeroLoop system has been successfully integrated with the federated RAWFIE infrastructure as a virtual testbed for experimenting with drones and other autonomous vehicles and has been used in other research projects, such as PV-Auto-Scout [135] which focuses on the automated inspection of photovoltaic parks via drones using IR-thermography, for evaluating the overall system functionality.

Next, Section 2.3.1 lists the supported functionality and Section 2.3.2 presents the design of the simulation environment.

2.3.1 Provided functionality

The supported functionality can be summarized as follows:

- Flight simulation of virtual drones (v-drones). This is done using the same autopilot software as in the real drone, configured to run in conjunction with a flight simulator in a SITL mode. The latter typically runs along the rest of the drone-specific software in an isolated, virtualized environment (VM/container).

- Virtual snapshot camera sensor. Software running on the v-drones can use a virtual camera device to capture photos during a mission. This is done through a high-level API that can invoke either a system-wide virtual camera service to retrieve aerial images based on the current position of the v-drone or sensor services provided by the mission visualization environment.

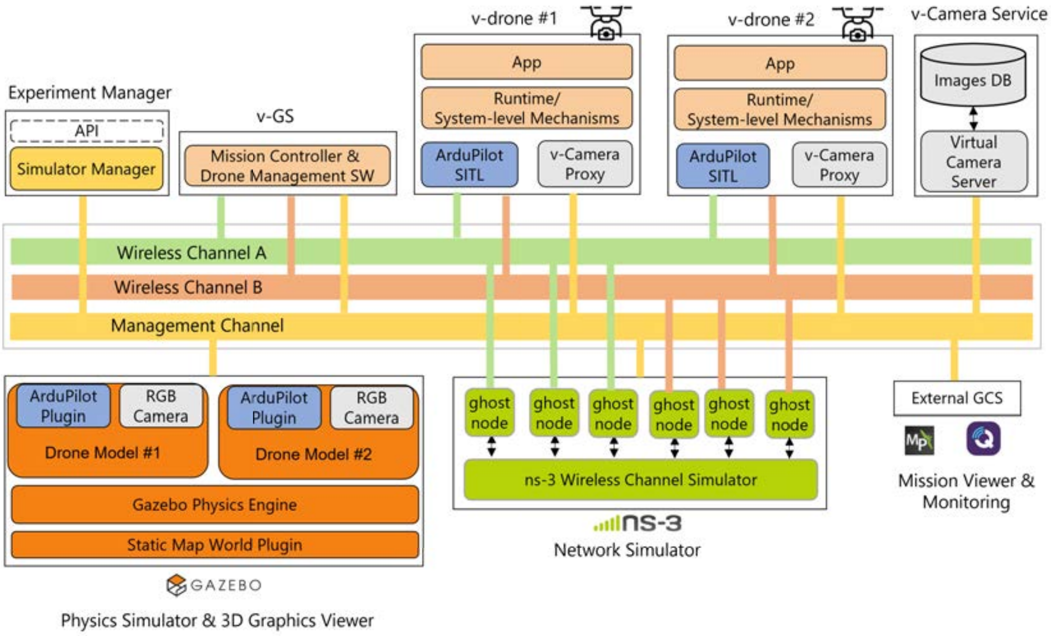


FIGURE 2.2: Architecture overview of the simulation environment

- Simulated wireless communication. Communication between v-drones and with other simulation entities, such as virtual ground stations (v-GS) or general-purpose virtual edge nodes, can take place through multiple virtual wireless channels simulating separate WiFi network domains or mimicking the behavior of other network technologies, e.g., 4G. This is done through a network simulator such as ns-3 [147] or by employing standard traffic control utilities on top of a Linux environment.
- Experiment visualization. The user can monitor the experiment and have a realistic view of the mission's progress either in 3D or in 2D. This is done through a graphics rendering engine provided by the flight simulator such as Gazebo [77], or through external ground control station programs such as Mission Planner [138].
- Programmatic experiment management. This is done through a structured API that can be used to perform the basic management operations to configure the required simulation entities, setup/tear down an experiment and retrieve various status information.

2.3.2 System overview

The simulation environment combines a number of different simulation techniques and takes advantage of mature virtualization technology in order to offer an integrated setup that allows experimentation with virtual versions of the various system entities, including the controller computer (v-GS) and the drones (v-drones) with their onboard camera. For better modularity and flexibility, each of these simulation entities can run in an isolated environment that can be either a Linux-based Virtual Machine (VM) or a container on top of a corresponding hypervisor. Figure 2.2 depicts the overall design/architecture while the main entities are presented below.

The experiment manager provides the API for performing all management tasks. The API is exposed through the zerorpc framework [128] and internally triggers interactions with the management agents that are included in all system entities (not shown in Figure 2.2 for brevity). This system-internal data exchange is performed via a dedicated management channel, which is implemented as an isolated bridge network.

The virtual ground station (v-GS) is the equivalent of a ground control computer and hosts the software related to the mission control and/or the drone management. Depending on the setup, it can communicate with the virtual drones through multiple simulated wireless networks.

The Gazebo simulator [77] is an integral part of the simulation environment and provides various functionalities, such as: (i) the description and configuration of the drone robotic models in Simulation Description Format (SDF) [50], (ii) the high-fidelity simulation of the drone's flight dynamics via the ODE physics engine [161], (iii) the production of sensor data for the drone's inertia measurement unit (IMU) as well as a visible light RGB camera, (iv) the three-dimensional visualization of the simulation environment via the OGRE graphics rendering engine [164], and (v) a ground plane with satellite imagery via the Static Map World plugin [131].

The virtual drone (v-drone) represents a simulated drone and hosts all application and system-level software. The autopilot uses the software-in-the-loop (SITL) ArduPilot simulation configuration [8], allowing to run exactly the same control code as in real drones without requiring real hardware. In this configuration, the autopilot during each control loop receives the data of the various flight sensors from the flight dynamics model (FDM) provided by the Gazebo simulator and then feeds backs the simulator with the commands it produces for controlling the motors. The robotic model used for the drone is based on the 3DR Iris quadcopter, which is a typical platform for a wide range of drone-based applications, while the interaction with the autopilot takes place via the ArduPilot Gazebo plugin [73]. The camera is accessed through the v-Camera Proxy that offers a pure Python interface, which is a simplified version of the picamera library [129] of the Raspberry Pi application development framework. Internally, depending on the specific scenario/configuration the v-Camera Proxy may interact with the v-Camera Service or the RGB camera of the Gazebo simulator.

The v-Camera Service simulates the functionality of a still snapshot camera, allowing the application software to take vertical aerial photos that depending on the experiment scenario can be either RGB or IR (thermal). Its implementation follows a service-oriented approach and can be accessed by many different drones concurrently. Its main components are: (i) a collection of aerial images (Images DB) covering the entire area of interest and (ii) a server program (Virtual Camera Server) that handles and serves requests. The images are stored as separate files in the local file system and contain appropriate meta-information about the geographical coordinates of the locations where they were received. The Virtual Camera Server at startup creates an in-memory index that holds the coordinates (latitude / longitude and altitude) for each such file, while during its operation, for each photo request it returns the one whose center is closer to the drone's 3D position. The camera control by the v-drone is performed via the Python Remote Objects frameworks Pyro4 [139], through the v-Camera Proxy which encapsulates in each such request the current position of the respective v-drone. More details about the virtual camera service can be found in [82, 81].

The alternative source of still images for the v-drone's camera comes from the drone's robotic model in Gazebo, which includes a visible light (RGB) camera. This camera retrieves the satellite imagery of its current field of view from the simulated ground plane and publishes the flow of images on a specific ROS topic [149]. In this case, at the v-drone side, the camera control is again performed through the v-Camera Proxy which subscribes to the respective ROS topic to capture still images.

The ns-3 network simulator [147] implements the virtual WiFi networks used for the communication between the v-drones and the v-GS. In particular, ns-3 creates a network of communicating simulated nodes, called ghost nodes, and models the IEEE 802.11 communication protocol standard at the physical and MAC layers, including the node mobility models. Each ghost node utilizes the ns-3 TapBridge device which is connected to each v-drone through a combination of network bridges and virtual network devices. The position of ghost nodes corresponding to drones is updated periodically through information propagated by the management agents of the respective v-drone entities. This interaction takes place on top of the management channel and follows a publish/subscribe scheme using the zeroMQ framework [180]. This way, each v-drone can access multiple separate wireless channels through different local network interfaces and the communication characteristics over each one of them are adapted dynamically as a function of the drone's mobility.

In addition, the simulation environment makes it possible to use additional tools that are useful for better visualization and monitoring of the status of a mission, especially in cases where multiple v-drones are involved. More specifically, external Ground Control Stations, such as Mission Planner [138] and QGroundControl [140], can be connected to the autopilots of the v-drones (in the same way this is done for real drones) to provide more detail on flight operation and provide a two-dimensional plane where all drones are shown simultaneously.

Depending on the experiment's scenario and the resource requirements of the involved entities, the simulation environment runs either on a typical server or on the cloud computing infrastructure of the ECE Department. In the former case, the simulation environment is installed natively in a Dell Precision Tower 5810 server equipped with an Intel Xeon CPU E5-1620 v4 at 3.50GHz and 16GB RAM [35], running a standard Linux distribution (Ubuntu 18.04). In the latter case, the entire simulation environment is encapsulated in a VM with 125 GB of RAM and 24 cores, deployed on top of the ESXi hypervisor that runs on a ProLiant BL460c Gen8 host equipped with 2 Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz and 256GB RAM.

2.4 Conclusion

In this chapter, we presented the experimental tools we use throughout this thesis for the evaluation of the mechanisms we design and develop. On the one hand, we have setup a hardware testbed that consists of typical platforms used in drone-based applications and can be configured for both lab and field experiments in a flexible way. On the other hand, we have developed a unified and modular simulation environment that combines mature simulation technologies for flight, sensing and networking to support easy and safe experimentation with virtual drones. The extensibility of the simulation environment is further demonstrated in Chapter 7, Section 7.4, where we discuss how it is used to introduce digital twin support in the context of a framework for managed drone applications.

Part I

Fault Tolerance Techniques in Coordinated Drone Missions

Chapter 3

System Model for Coordinated Drone Missions

In this chapter, at first, we describe in detail the system model we use for coordinated multi-drone missions and then we introduce the main extensions that enable our fault tolerance mechanisms. The system model assumptions that are specific to each one of the proposed mechanisms are presented at the respective chapters.

3.1 Basic model

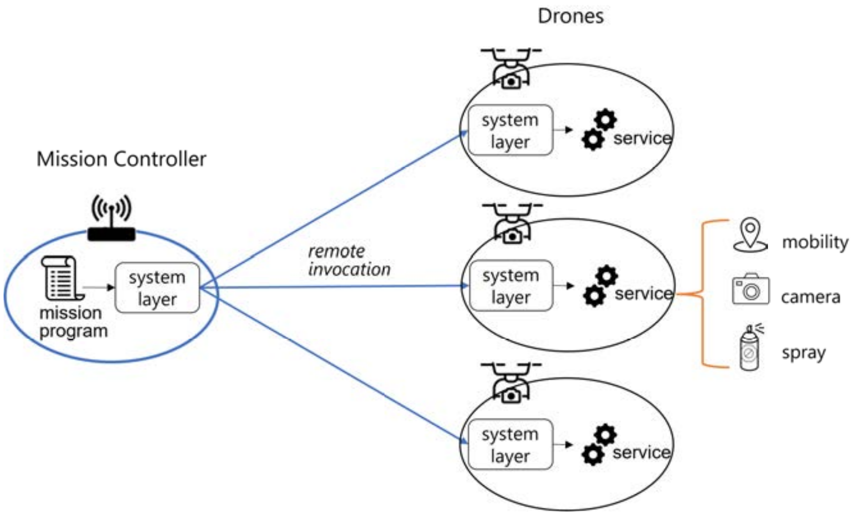


FIGURE 3.1: System model for coordinated drone missions

Figure 3.1 presents the system model for coordinated drone missions. We model drones as powerful sensor/actuator *nodes* with movement capability. We assume that each node comes with substantial navigation and obstacle-avoidance capability and executes all critical control loops locally using an autopilot that runs on an embedded real-time platform. The high-level coordination of the mission is performed by a distinguished entity, the *mission controller*. The mission controller runs the *mission program*, an application that collects state information from the nodes, processes and analyzes this information, takes high-level coordination decisions, and issues corresponding commands to the nodes according to the mission objectives. Thus, the mission controller effectively acts as the master that takes all high-level coordination decisions, while the nodes act as slaves that follow the commands of the master. Nodes do not need to communicate with each other for this purpose.

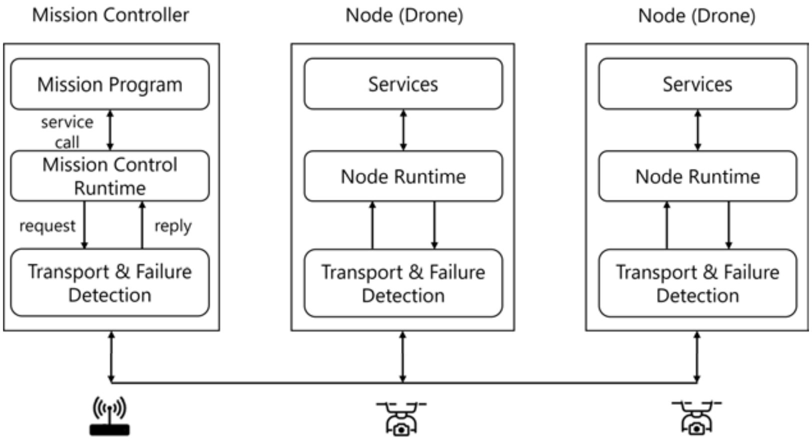


FIGURE 3.2: Information flow between the mission controller and the nodes (drones)

In the spirit of service-oriented computing [42], the sensing, actuation and mobility/navigation capabilities of the nodes are exposed in a structured way, through corresponding *services* with well-defined remote interfaces. These interfaces consist of function calls that can be used to retrieve state information from the drone as well as to issue commands prompting it to perform some action that may concern a movement or an onboard actuator.

Figure 3.2 illustrates the information flow between the main system entities. The interaction between the mission program and the node services is implemented in a transparent way, through suitable runtime support. Among other things, the runtime is responsible for implementing the remote service invocation in the spirit of remote procedure calls (RPCs) [122]. Each service call translates to a corresponding request that is sent to the target node. In turn, the node processes the request and sends back the reply. This can be supported using suitable RPC support or be implemented directly on top of a reliable transport service, such as TCP/IP.

The nature of the *operation* that is requested via a service call can be synchronous or asynchronous. Synchronous operations are performed immediately, and the outcome is encoded in the return value of the call. Asynchronous operations typically take a longer time to be performed. In this case, the return value merely acknowledges the fact that the node will try to perform the requested operation (which may be completed with some delay). The progress of asynchronous operations can be monitored by issuing service calls that return the desired state information. A typical example is to perform a call that instructs the node to go to a certain location and then periodically retrieve the node's current location via a different call to monitor movement progress. Unlike asynchronous procedure calls, in our case, the remote service calls are always synchronous and block the caller (i.e., mission program) until the reply is returned, irrespective of whether the operation they trigger in the node is synchronous or asynchronous.

For node failures, we assume the fail-stop model: a node either functions properly or stops. This is according to standard practice for embedded systems, which internally employ redundancy and/or secure state estimation techniques [54] [44] so that they are at least able to detect software/hardware failures of sensors/actuators as well as adversarial attacks on them. Moreover, in the case of autonomous vehicles, severe malfunctions almost inevitably lead to actual crashes [162].

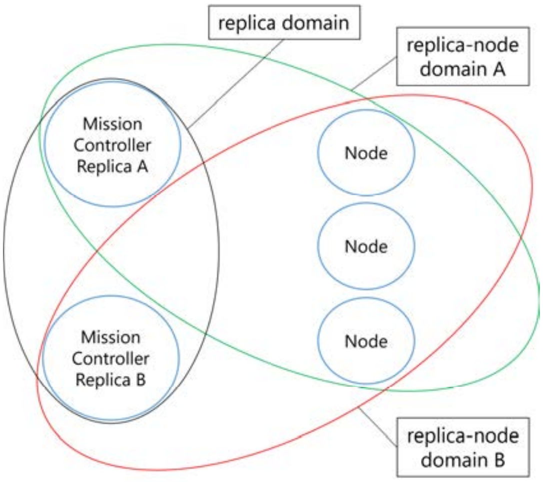


FIGURE 3.3: Organization of the communication for the active replication of the mission controller

We assume that remote service invocations have at-most-once semantics [105] and that node failures are detected at the transport/RPC layer. If a service call returns normally, then the mission program knows that the node has processed the request exactly once. Else, if the call returns with an error indicating a node failure, the mission program does not know whether the node managed to handle the request, but it is certain that the node did not execute the call more than once. This is particularly important for calls that lead to critical actuation operations, which may be non-idempotent [31].

The mission controller may also fail during mission execution. From a traditional RPC perspective, this corresponds to a client failure, which is typically dealt with by garbage-collecting orphan calls at the server, using a suitable mechanism such as extermination or reincarnation [122]. In our case, when a node detects a failure of the mission controller, it enters a fail-safe state in which it remains until a human operator takes over control manually.

In the presented system model, node failures and failed service invocations are not fatal, as it may be possible to proceed with the mission by employing additional nodes and/or redistributing the tasks that need to be performed among the remaining nodes. However, the mission controller constitutes a single point of failure for the execution of the mission.

3.2 Extended model

To tolerate failures of the mission controller in a transparent way, we extend the basic system model and propose an active replication approach where the (same) mission program is actively executed by multiple instances of the mission controller. We refer to each instance of the mission controller as a *replica*. Figure 3.3 shows the organization of the communication that takes place between the mission controller replicas and the nodes, as this is assumed in our approach.

Each replica interacts with the nodes within a separate/private communication domain. We refer to this as a *replica-node* domain. From a network layer perspective, replica-node domains can be implemented as VPN over the Internet, or via dedicated wireless telecommunication links. To avoid crosstalk between the domains and to increase reliability against jamming attacks, each domain could use a different radio frequency, possibly even a different radio technology for increased reliability.

Replicas interact with each other using a separate communication domain, the *replica* domain. Unlike the replica-node domains, the replica domain could be implemented using tethered-only networking technology (if the replicas are stationary).

Note that active replication is traditionally used in computer systems for servers/ services that provide critical functionality [155]. In contrast, in our case the most critical system component is the mission controller, which according to the client-server model semantics acts as a client by invoking the services provided by the nodes.

Both fault tolerance mechanisms we propose in this thesis, presented in Chapters 4 and 5, assume this extended system model. Thus, in the respective sections we focus on the assumptions that are specific to each approach.

Chapter 4

Tolerance of Fail-Stop Failures Using Active Replication

4.1 Contributions and Outline

In this chapter, we investigate how to support the active replication of the mission controller in centrally coordinated drone missions to tolerate fail-stop failures, while addressing all the related correctness/consistency issues. We consider such a research direction as an important step towards making this kind of systems more dependable, which would benefit a wide range of applications by increasing their robustness and achieving a truly autonomic operation.

Our main contributions are the following:

- We present an approach for supporting the active replication of the mission controller in coordinated teams of autonomous vehicles like drones.
- We define the properties that should be satisfied to ensure consistency and highlight the issues that need to be addressed.
- We present, in detail, a system-level mechanism that tackles the problem for deterministic mission programs, and briefly discuss how to deal with non-deterministic mission programs.
- We capture the overheads of a concrete implementation of the proposed solution, analytically as well as experimentally.

The rest of the chapter is organized as follows. Section 4.2 lists the key assumptions of the system model we use as a baseline for our work. Section 4.3 defines the desired consistency properties, identifies the issues that arise when employing an active replication approach for the mission controller and presents solutions for deterministic and non-deterministic mission programs. Section 4.4 discusses a concrete implementation of the proposed solutions for a programming environment that supports multi-drone applications. Finally, Section 4.5 gives an analytical cost model for the communication overhead of the proposed approach and presents the results of indicative tests that have been performed using our prototype implementation on top of a suitable simulation environment.

4.2 System Model

In our system model, drones are modeled as *nodes* that expose the available sensing, actuation and mobility resources in the form of *services* that are being invoked remotely by a *mission program* that describes the high-level mission objectives. The same mission program is actively executed by multiple *mission controller replicas*, and each replica invokes the nodes inside a separate *replica-node communication domain*. Replicas communicate through the *replica domain*, whereas nodes do not need to interact with each other for the purposes of the mission execution. For more details, refer to Chapter 3.

The key assumptions are summarized as follows:

- Remote node service invocations have at-most-once semantics [105].
- Node failures follow the fail-stop model, i.e., a failed node stops operating.
- Node failures are detected by the replicas at the system's transport/RPC layer.
- Mission controller replicas can experience fail-stop failures.
- Replicas interact using reliable group communication, such as reliable multicast [26].
- Replicas' communication comes with a reliable failure detection mechanism.
- Mission programs can be either deterministic or may include non-deterministic operations.

The fact that the replicas reliable group communication provides failure detection functionality ensures that if a replica fails then the failure will be announced to all other working replicas after the last message that was sent by the failed replica is delivered. The fault tolerance mechanism we propose in the sequel works with simple FIFO delivery, without requiring more advanced/costly causal or total ordering [99].

4.3 Fault Tolerance Mechanisms

In this section, we present the properties that summarize the desired functionality and next we present a mechanism that ensures them for deterministic mission program executions. We start by considering executions without node failures and then discuss the extensions needed to deal with node failures. Finally, we briefly discuss how to support the execution of non-deterministic mission programs.

4.3.1 Replication properties

In the traditional active replication scheme, several properties have been defined to capture the consistency requirements for a deterministic replicated service [34]. In our case, where the replicated entity is the mission controller, we capture the desired consistency and functionality via two properties:

- **Uniform Request Agreement.** All working replicas issue the same requests in the same order.
- **Uniform Reply Integrity.** All replicas that issue the same request, will receive the same reply.

Figure 4.1 gives an overview of the entities, the mechanisms that we introduce and the interactions that take place in our system model. For mission applications with a deterministic execution flow, *Uniform Request Agreement* is ensured by default. For

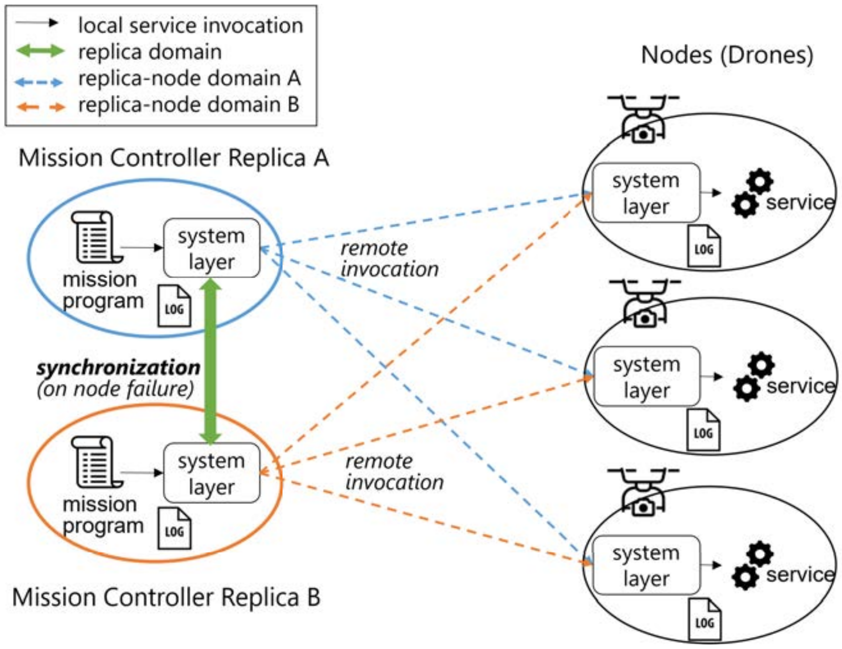


FIGURE 4.1: Overview of the system entities, their interactions and the introduced mechanisms

non-deterministic execution, appropriate replica coordination is needed, as we discuss in subsection 4.3.8. In the absence of node failures, *Uniform Reply Integrity* can be ensured by uniquely identifying the requests and keeping track of the executed requests as we show in subsection 4.3.3. However, if a node fails then the situation becomes more complicated, and synchronization is required. We discuss and address this issue in subsection 4.3.4.

4.3.2 Fault-free operation

Assuming deterministic mission program execution, the replicas of the mission controller can execute the mission program in parallel to each other, without any synchronization (as long as there are no node failures, which are discussed in the sequel). This has the advantage of reduced communication overhead and faster mission progress. Due to this decoupled parallel execution, the execution of the mission program at some replicas may lag behind other replicas. We refer to a replica as *fast* if it issues a service call that has not yet been issued by another replica. A replica is called *slow* if it issues a call that has already been issued by some replica(s).

Let replica r_k keep a sequence number seq_k that is increased each time it performs a service call to a node. Replica r_f is fast if $seq_f \geq seq_k, \forall k$, whereas r_s is slow if $\exists k \mid seq_s \leq seq_k$. In the same vein, r_f is the slowest if $seq_s \leq seq_k, \forall k$. Of course, all replicas might issue requests at the same speed, in which case they are all equally fast/slow.

4.3.3 Duplicate requests handling

Since the mission is deterministic, a node receives the same service request multiple times, once from each replica. However, each service call should be executed at most once.

To deal with such duplicate service calls, each node keeps a log of the requests received and the replies that were produced in return. Let $log[pos].seq$ and $log[pos].req$ be the sequence number and respectively a hash of the request at position pos in the log and let $log[pos].rpl$ be the corresponding reply. Also, let pos_k be the log position for the last request of r_k. Finally, let pos_last be the position of the last log entry, which corresponds to the last new (not duplicate) request received.

Algorithm 1 shows how a node handles a service call request coming from a replica of the mission controller, while Algorithm 2 shows how a replica handles the reply that is sent from a node in response to a previously issued request.

When a node receives the next request req_k from r_k, it checks whether $pos_k = pos\_last$. If so, this is a new request, thus the respective service call is executed, the log pointer of r_k is incremented $pos_k = pos_k + 1$ and the request is appended to the log together with the reply that is sent back. If $pos_k < pos\_last$, the request of r_k has already been issued by another replica and it is checked whether it is identical to the one stored in the next position of the log, $req_k = log[pos_k + 1].req$. If so, the log pointer is incremented, $pos_k = pos_k + 1$ and the corresponding reply $log[pos_k].rpl$ is retrieved from the log and is sent to the replica. Else, the node sends a reply indicating that this is an unexpected request, so that the mission controller can act accordingly, e.g., notify the mission program via a corresponding runtime exception. This can only occur in an exceptional scenario, discussed in Section 4.3.5.

Algorithm 1 Handling incoming service calls at the node

```
$log \leftarrow \emptyset$                                        ▷ request/reply log
$pos\_last \leftarrow 0$                                          ▷ last log position
$seq\_min \leftarrow 0$                ▷ smallest sequence number over all replicas
for each replica $r_k$ do
    $pos_k \leftarrow 0$                              ▷ position of last request of $r_k$
end for

upon receiving $\langle REQUEST, seq_k, req_k \rangle$ from $r_k$ do
    if $pos_k = pos\_last$ then                                       ▷ new request
        $rpl_k \leftarrow$ execute ($req_k$)
        $pos_k \leftarrow pos_k + 1$
        $pos\_last \leftarrow pos_k$
        append($log, (seq_k, req_k, rpl_k)$)
        $seq\_min \leftarrow$ min($log[pos_k].seq | \forall k$)
        send $\langle REPLY, seq_k, rpl_k, seq\_min \rangle$ to $r_k$
    else if $pos_k < pos\_last$ then                                  ▷ old request
        if $req_k = log[pos_k + 1].req$ then
            $pos_k \leftarrow pos_k + 1$
            $seq\_min \leftarrow$ min($log[pos_k].seq | \forall k$)
            send $\langle REPLY, seq_k, log[pos_k].rpl, seq\_min \rangle$ to $r_k$
        else                                       ▷ mismatch with logged request
            send $\langle UNEXPECTED\_REQ, seq_k \rangle$ to $r_k$
        end if
    end if
end
```

Algorithm 2 Handling node replies at the replica

```
$log \leftarrow \emptyset$
$seq\_slow \leftarrow 0$                                ▷ sequence number of slowest replica(s)

upon receiving $\langle REPLY, seq, rpl, seq\_min \rangle$ from $n_i$ do
    append $(log, (seq, n_i, rpl))$
    $seq\_slow \leftarrow \max(seq\_slow, seq\_min)$
    return $rpl$
end

upon receiving $\langle UNEXPECTED\_REQ, seq \rangle$ from $n_i$ do
    return $UnexpectedReqError$                         ▷ notify problematic situation
end
```

4.3.4 Node failures

When a node fails, it is no longer guaranteed that all replicas will continue their execution in the same way, even if the mission program is deterministic. This is because some slow replicas will not be able to receive any replies from the failed node, and thus will notify the mission program about the node failure at an earlier point of execution compared to a faster replica. As a result, the mission program may take a different execution path than the one that was followed by the fast replica(s), which may have successfully invoked the node before it failed. To address the problem, replicas have to synchronize to ensure that they will all continue the execution of the mission program in the same way. Note that the node failure may be discovered by any replica irrespectively of it being fast or slow.

To support the required synchronization, every replica r_k also maintains a log with entries for the service calls it has performed to the nodes. Let $log[pos].seq$, $log[pos].n$ and $log[pos].rpl$ denote the call sequence number, the target node and the reply that was received from the node, respectively. Also, every node n_i records the current call sequence number of each replica and updates the minimum value, which is included in the replies sent to the replicas (see Algorithm 1, lines 13-14 and 18-19). Based on this information, in turn, each replica r_k maintains a conservative lower bound for the call sequence number of the slowest among all replica(s), let this be seq_slow (see Algorithm 2).

When replica r_k detects that node n_f has failed, it stops the execution of the mission program and enters a special synchronization state. After recording the failure, it sends to all replicas, via reliable multicast, a synchronization message that includes n_f and seq_k. The message also includes the log entries for all the calls r_k has issued to n_f that may have not yet been performed by some slower replica(s), i.e., all entries where $log[pos].n = n_f \land seq\_slow < log[pos].seq \leq seq_k$.

When a replica receives a synchronization message for node n_f for which it has not yet detected the failure, it acts as if it had just detected the failure of n_f (as discussed above). Also, for every synchronization message received, replicas add to their log any missing entries for the failed node n_f, update the sequence number of the last call that was issued by any replica to n_f, let this be $seq\_last[f]$, and update the sequence number of the slowest replica seq_slow.

The size of the synchronization messages exchanged between the replicas can be reduced using a simple optimization. Namely, if a replica learns about a node failure from another replica r_k, it suffices to include in its own synchronization message only the log entries for the failed node that are not already found in the message of r_k (which will be received by all replicas, thanks to the reliable multicast functionality).

Algorithm 3 Replica synchronization for a node failure

```
$state \leftarrow normal$
**for each** node $n_i$ **do**
    $failed[i] \leftarrow false$        ▷ own failure detection flag for $n_i$
    $sync[i] \leftarrow 0$        ▷ nof sync messages received for the failure of $n_i$
    $seq\_last[i] \leftarrow 0$        ▷ sequence number of last call to $n_i$ in the log
**end for**

**upon** detecting failure of node $n_f$ **do**
    **if** $failed[f] = false$ **then**
        $state \leftarrow sync$
        $failed[f] \leftarrow true$
        $seq\_last[f] \leftarrow$ getLastReqSeqno$(log, n_f)$
        $log_f \leftarrow$ getLogEntries$(log, n_f, seq\_slow, seq)$
        send $\langle SYNC, n_f, seq, log_f \rangle$ via RM
        **if** $(failed[i] = false) \vee (sync[i] =$ nofReplicas()$), \forall n_i$ **then**
            $state \leftarrow normal$
        **end if**
    **end if**
**end**

**upon** receiving $\langle SYNC, n_f, seq_k, log_k \rangle$ from $r_k$ **do**
    $sync[f] \leftarrow sync[f] + 1$
    $seq\_slow = min(seq\_slow, seq_k)$
    **if** $seq < seq_k$ **then**
        appendLogEntries$(log, log_k, seq, seq_k)$
        $seq\_last[f] \leftarrow$ getLastReqSeqno$(log, n_f)$
    **end if**
    **if** $failed[f] = false$ **then**
        $state \leftarrow sync$
        $failed[f] \leftarrow true$
        $seq\_last[f] \leftarrow$ getLastReqSeqno$(log, n_f)$
        **if** $seq > seq_k$ **then**
            $log_f \leftarrow$ getLogEntries$(log, n_f, seq_k, seq)$
        **else**
            $log_f \leftarrow \emptyset$
        **end if**
        send $\langle SYNC, n_f, seq, log_f \rangle$ via RM
    **end if**
    **if** $(failed[i] = false) \vee (sync[i] =$ nofReplicas()$), \forall n_i$ **then**
        $state \leftarrow normal$
    **end if**
**end**
```

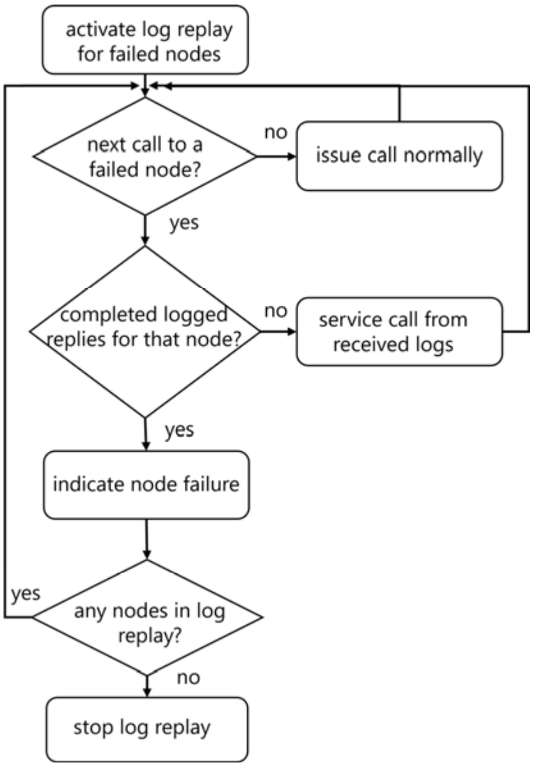

activate log replay for failed nodes
next call to a failed node?
no
issue call normally
yes
completed logged replies for that node?
no
service call from received logs
yes
indicate node failure
yes
any nodes in log replay?
no
stop log replay

FIGURE 4.2: Flowchart of extended service call process

Note that it is straightforward to handle the case where different replicas concurrently discover the failure of different nodes. In this case, all replicas remain in the synchronization state and repeat the process for the additional failed node(s) before resuming the mission program. Replica r_k remains in the synchronization state until it receives a synchronization message from every other replica. Then, it reverts to the normal state and resumes the execution of the mission program.

Algorithm 3 shows the synchronization between replicas when a node failure is detected. This includes the optimization that avoids sending superfluous log entries when a replica learns about a node failure as a side-effect of receiving the synchronization message of a faster or equally fast replica. It also handles the case where different node failures are detected simultaneously, in which case, multiple synchronization rounds are performed in parallel, one round for each node.

Finally, an extended service call process is activated as illustrated in Figure 4.2. When the mission program issues a service call, before sending a request to the target node n_i, the replica checks whether it has recorded a failure of n_i. If not, the node is invoked as usual. Else, it is checked whether the log contains the reply for this request ($seq \leq seq\_last[i]$), in which case the reply $log[pos].rpl | log[pos].seq = seq$ is fetched from the log and is returned to the mission program. If the log does not contain the reply, an error is returned to the mission program, indicating that n_i has failed (the call sequence number is not incremented in this case). Algorithm 4 gives a high-level description of this process.

Algorithm 4 Extended service call process at the replica

```
when invoking $n_i$ with request $req$ do
    $seq \leftarrow seq + 1$
    if $failed[i] = false$ then                ▷ issue request as usual
        send $\langle REQUEST, seq, req \rangle$ to $n_i$
    else if $seq \leq seq\_last[i]$ then       ▷ get reply from the log
        return getLogReply($log, seq$)
    else
        $seq \leftarrow seq - 1$
        return $NodeFailureError$              ▷ indicate node failure
    end if
end
```

4.3.5 Replica failures

When a node detects the failure of a replica r_f, it simply removes r_f from the set of replicas from which it expects to receive requests. It may also garbage collect all internal data structures concerning r_f. Nodes will enter the fail-safe state only when all replicas of the mission controller fail.

When a replica detects the failure of another replica r_f, it removes r_f from the set of working replicas in order for this replica to be excluded from subsequent communication/synchronization rounds. Note that if a replica fails during a synchronization round, thanks to the reliable multicast functionality, its message will either be received by all working replicas or by none of them, so this will not affect the outcome.

However, there is a corner case where the failure of a replica will cause a problem, namely if node n_f fails and there is a single fast replica r_f, which is the only one that has performed the most recent service call(s) to n_f, and r_f fails too, before it manages to send its own synchronization message with the missing log entries for n_f. In this case, the remaining replicas cannot recover these entries and there is no safe way for them to deduce that this is a problematic situation. Thus, they continue with the execution of the mission program as usual. However, this may lead to a different execution path from the one followed by the fast replica r_f. This situation is detected when a replica receives a reply indicating that its request is unexpected (see Section 4.3.3, Algorithm 1 line 20 and Algorithm 2 line 9). Then, the mission program is notified in order to handle the problem (e.g., set the nodes in fail-safe mode and retrieve information from them to assess the current situation). If the prospect of such a discontinuity is not acceptable, one has to adopt the more conservative approach for supporting non-deterministic execution, discussed in Section 4.3.8.

4.3.6 Garbage collection of log entries

The logs of the nodes and the replicas cannot grow indefinitely. Fortunately, both logs can be garbage-collected in a straightforward way, without any additional communication.

Nodes can remove a log entry once a corresponding service call request is received from every working replica. More specifically, all entries with $log[pos].seq \leq seq\_min$ can be safely removed from the log. This way, the log of a node only contains entries for the service calls that have not yet been performed by the slowest replicas.

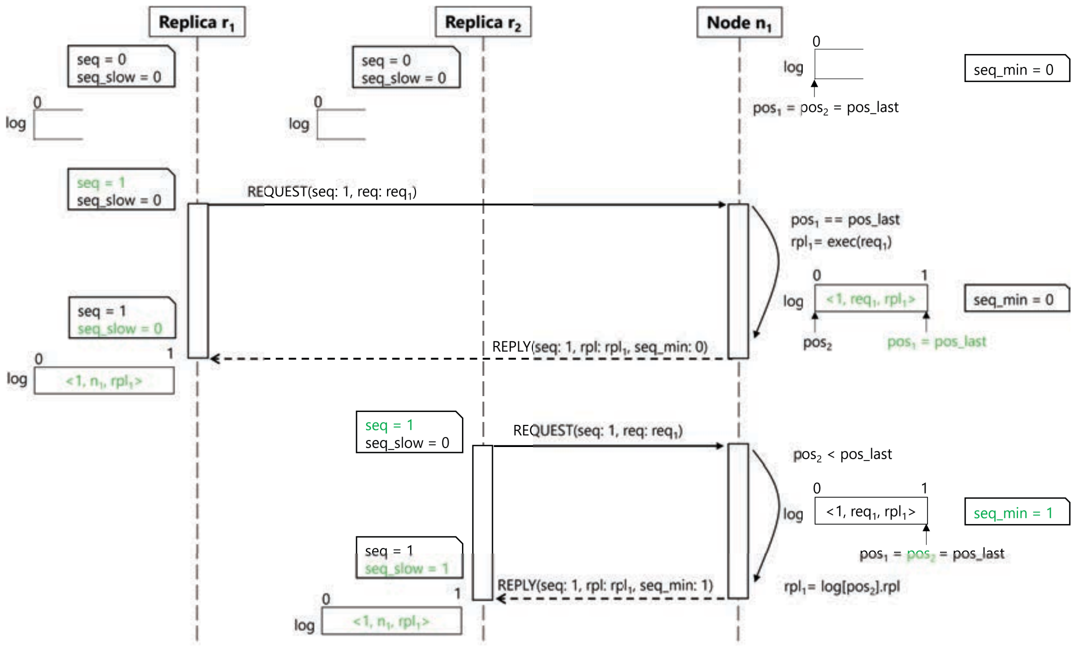


FIGURE 4.3: Invocation of the same node service by two replicas

Along the same lines, a replica can truncate its log up to the entry where $log[pos].seq = seq\_slow$, which represents a conservative lower bound for the last call that has been performed by the slower replicas. Recall that seq_slow is calculated independently by each replica, based on the information that nodes append to their replies. Also note that during the synchronization phase, all replicas can update seq_slow to accurately reflect the smallest call sequence number among all replicas, and thus can safely truncate their log accordingly.

4.3.7 Indicative operation sequence diagrams

To provide a better overview of our approach, we present the operation of our mechanisms for two scenarios regarding the two most basic operations, namely the service invocation and the handling of node failures.

Figure 4.3 illustrates how a service invocation is handled at the node and at the replicas. It shows two replicas, r_1 and r_2, sending the same request to node n_1. The green color indicates changes in the state of the respective entity. When the request from r_1 is received, since it is a new request, the service call is executed and inserted in the node's log. On the other hand, as the request from r_2 is identified as a duplicate it is served from the log. At the replicas' side, the received replies are also logged. Note that n_1 keeps the smallest sequence number of the requests received from all the replicas (seq_min) and this information is also included in the replies it returns. In turn, this can be used by both replicas and the node to truncate their logs when needed. For instance, in the particular scenario assuming that there are no other replicas r_2 and n_1 can garbage collect their logs.

Figure 4.4 presents the interactions that take place during a node failure. As above, the green color indicates changes in the state. Also, the numbers inside the log entries indicate the sequence number of the respective requests. When replica r_1 detects the failure of n_1, it enters the synchronization state and sends to r_2 a $SYNC$ message via reliable multicast to inform it for the failure event. The message also

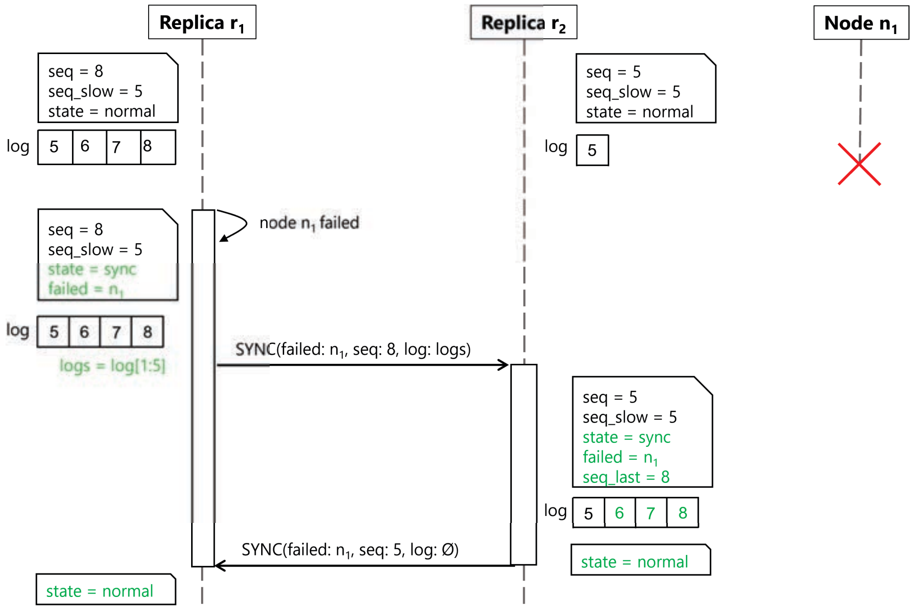


FIGURE 4.4: Replicas synchronization on node failure

carries the logged replies from n_1 corresponding to requests that may have not been yet performed by the slower replicas (from request sequence number 6 to 8). Replica r_2 enters the synchronization state upon receiving the message, appends its log with any missing entries and since it is slower than r_1 sends the *SYNC* message without including any log entries. Once, the replicas have received synchronization messages from all other replicas they resume the mission program execution. Note that replica r_2 will announce the node failure to the mission program once it has completed servicing its requests to n_1 from the logged replies.

4.3.8 Non-deterministic execution

The above approach will only work if the mission program is deterministic. But requiring the mission logic to be deterministic can be restrictive. Moreover, as discussed above, there is a corner case that may lead to mission discontinuation.

Non-deterministic execution can be supported by adopting a semi-active replication approach [176]. Adapted to our context, this works as follows. When a non-deterministic operation is encountered in the mission program, the replicas pause their execution in a barrier-like manner. Then, a distinguished replica, the *leader*, executes the non-deterministic part of the mission program and when done transfers the local execution state to the *follower* replicas (again, this can be done using simple reliable FIFO multicast). The followers, in turn, update their state and resume the execution of the mission after the endpoint of the non-deterministic execution.

If the leader fails, a new one is elected. Given that any replica can assume this role, the election can be done based on trivial information that is already available locally at each replica. For example, the role of the leader can be assigned to the replica with the largest identifier.

The nodes keep the same log structure as discussed above to avoid executing duplicate service calls during the execution of deterministic sections. In addition, the requests issued by the leader during a non-deterministic section are flagged so that the next request of a follower replica, when it resumes execution after a non-deterministic section, will not be detected as unexpected (there will be a gap in the sequence numbers of the previous and the next request of the replica, before and respectively after the non-deterministic section). Note that if the leader fails while executing a non-deterministic section, the new leader that takes over will perform the last service call that may have already been performed by the previous leader before it failed. Such duplicates can be detected and handled by the node as usual.

As will be shown in Section 4.5, this mode of operation comes at a significant cost. Therefore, it is important for missions to be designed so that they have few, well-defined non-deterministic sections. The worst case is for the entire mission program to be non-deterministic, in which case the synchronization between the leader and the followers has to be performed at every service call towards a node.

4.4 Implementation

We have implemented the proposed replication approach in a suitably extended version of the TeCoLa programming framework [85]. TeCoLa is a Python-based middleware, designed to facilitate the high-level coordination of dynamic and heterogeneous robotic teams. Its design follows the system model discussed in Section 4.2.

For the interaction between the mission controller and the nodes we employ GCBRR, a reliable 1-N request-reply transport with group management capabilities [83]. GCBRR is designed for channels with physical multicast capability and supports 1-to-N request-reply exchanges with the minimum number of message transmissions, at low latency and without any contention among the communicating parties. Each service call performed by the mission programs is mapped, behind the scenes, to a corresponding request-reply interaction.

The replicas of the mission controller are specified in a configuration file, which is available at all replicas and nodes. This way, all replicas know each other, and nodes know from which replicas to expect service calls, without requiring some discovery infrastructure/protocol. The current implementation does not support the dynamic addition of replicas at runtime. Reliable multicast communication between the replicas of the mission controller is implemented using the JGroups toolkit [66]. We configure JGroups to use UDP/IP as transport in order to exploit IP multicasting, to combine FD and FD_SOCK options to achieve solid failure detection among the replicas and to employ the NAKACK2 protocol for the reliable FIFO delivery of multicast messages using negative acks.

To support non-deterministic mission programs, a replica must be able to save and restore the state of the mission program. In our implementation, this is done using the DMTCP framework [6]. To accelerate prototyping, we save the entire state of the process that runs the mission program in a brute-force way, without attempting a more elaborate integration with the TeCoLa environment (which might allow a more selective recording of the absolutely crucial state information). To reduce the size of the images, we perform incremental checkpointing using the HBICT module [60], which works seamlessly with DMTCP.

4.5 Evaluation

In this section we discuss the communication overhead of the proposed active replication approach. On the one hand, we give analytical estimations for the main components of the mechanisms. On the other hand, we record the overhead of the current implementation in TeCoLa and compare them with the analytical estimates. Our analysis assumes that a node handles incoming requests in a serialized/FIFO manner, thus, the handling of the next request starts after the handling of the previous request is completed (as done in the implementation). Also, we assume that the replica-node domains are isolated and do not crosstalk/interfere with each other.

4.5.1 Experimental setup

The experimental measurements are performed using the simulation environment we have developed and presented in chapter 2, section 2.3, which allows us to test mission software by running experiments with several virtual unmanned aerial vehicles (v-drones) that can be controlled from a virtual ground station (v-GS). In this case, the v-drones and the v-GS run the TeCoLa software stack.

We introduce multiple v-GS, each running a replica of the TeCoLa mission controller. The replica-node domains are implemented as separate wireless WiFi networks, with the simulation support of ns-3. The WiFi rate is set at the basic rate for multicasts, 1 Mbps. The replica domain is implemented as a wired Ethernet network with a rate of 210 Mbps.

The mission program that runs on top of TeCoLa, performs a series of dummy service calls. We set the requests and replies either so that their payload fully occupies the maximum packet payload, or so that their payload is empty, in which case the respective packets only carry the basic protocol headers. We can also set the amount of processing that is performed by the node for each such call. In our experiments, we vary the number of replicas used for the mission controller. Note that the overhead of the proposed replication scheme does not depend on the number of nodes employed in the mission.

4.5.2 Service call delay in deterministic execution

First, we investigate the end-to-end delay of a service call for deterministic execution scenarios, for the typical case where the target node is alive. This can be expressed as $T_{call} = T_{rtt} + T_{proc}$. The round-trip time T_{rtt} is the time it takes to send the request and receive the reply over the communication channel of the node-replica domain. For a fast replica, T_{proc} equals the time needed by the node to process the request. For a slow replica, $T_{proc} = 0$ if the node is idle, as the reply is directly fetched from the log. If, however, the node is busy processing a request of a fast replica, in the worst case, T_{proc} will be equal to the respective processing delay.

In a first set of experiments, we measure T_{call} for the case where there are no node failures. We use three different service calls, which perform a brute-force primality test, with a processing time T_{proc} of 1, 2 and 3 seconds, respectively. Each request and reply carry the maximum packet payload for WiFi, 1500 bytes, yielding a T_{rtt} of 26 milliseconds at the WiFi channel rate of 1 Mbps. We employ two replicas of the mission controller (A and B), running a mission program that performs these service calls, one after the other. We experiment with three execution scenarios. In the *parallel* execution, both replicas perform the same call practically at the same

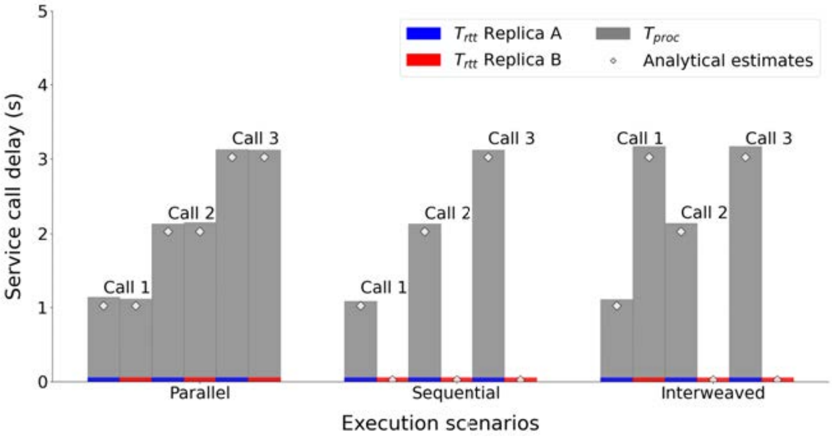


FIGURE 4.5: Service call delay in the absence of node failures

time. In the *sequential* execution, replica A performs the first service call and once it returns, then replica B proceeds to perform the same call. Replica A performs the next service call right after the completion of the previous call at replica B. In the *interweaved* execution, replica A is two calls ahead of replica B and replica B performs the first service call when replica A already performs the last call.

Figure 4.5 reports the analytical and experimental results. In the parallel execution scenario, the call delays at both replicas are the same and equal to the call delay when using a single replica. In the sequential execution, replica B experiences significantly lower service call delays, which basically amounts to the round-trip time. This is because these calls have already been processed by the node due to the calls performed by replica A, so the node simply returns the replies from the log. Finally, in the interweaved execution, for the first call, the slower replica B experiences the same delay as the faster replica A for the third (and more time-consuming) call, while the rest of the calls execute very fast, like in the sequential execution scenario.

It is important to stress that the experimentally measured delays are close to the ones estimated analytically. In general, the service call delay is strictly bounded by the processing time of the most time-consuming service call. Also note that any additional replica(s) would experience a call delay within the lower and upper bounds reported here, depending on the time of invocation with respect to the faster replica.

4.5.3 Replica synchronization delay on node failures

Once a node failure has been detected and the replica synchronization has been completed, all subsequent calls of the mission program to that node are handled based on the local log of the mission controller, without any communication, thus T_{call} is negligible. Therefore, we focus on the overhead of the synchronization between the replicas.

We analytically estimate $T_{sync} = N \times T_{log}$, where N is the number of replicas and T_{log} is the time it takes for a replica to send its own log entries to all other replicas via reliable multicast. This, in turn, can be expressed as $T_{log} = LogS/MaxP \times T_{rm}$, where $LogS$ is the total size of the log entries to send, $MaxP$ is the maximum packet payload for the underlying network and T_{rm} is the time it takes to send a reliable multicast message that occupies a full packet. With JGroups configured to exploit IP multicasting and negative acknowledgements, each reliable multicast roughly translates to a

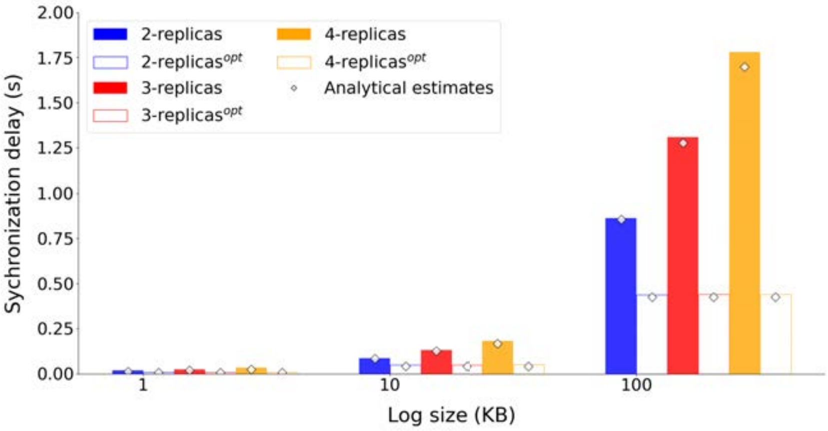


FIGURE 4.6: Replica synchronization delay on node failure

single packet transmission, yielding a T_{rm} of about 6 milliseconds for a fully loaded packet of 1500 bytes over the 210 Mbps Ethernet network of the replica domain.

In a second series of experiments, we measure T_{sync} for 2, 3 and 4 replicas running a mission program that periodically invokes a node. We also vary the total size of the log entries that need to be exchanged between the replicas ($LogS$), from 1 KB, 10 KB up to 100 KB. Note that $LogS$ depends on the number of service calls performed by the fastest replica to the node (before it failed) that have *not* yet been performed by the slowest replica, as well as on the size of the node's replies to these calls. But what actually matters is the total size of this log information.

Figure 4.6 shows the results (average over 20 runs; there is no significant deviation) for the naive and optimized version of the synchronization protocol (see Section 4.3.4). The analytical estimates are, as before, very close to the measured delays. We observe that the synchronization delay increases linearly to the size of the replica logs and the number of replicas. Notably, the optimized version reduces the synchronization delay significantly, which practically becomes constant irrespectively of the number of replicas; for every additional replica the increase is lower than 0.01%. This becomes clearly visible for larger log sizes. The reason is that only the replica that detects the failure first, includes in its synchronization message the log entries for the failed node, while all other replicas do not re-send the same entries and thus generate very small synchronization messages.

4.5.4 Service call delay in non-deterministic execution

The delay of a service call in the non-deterministic execution mode can be expressed as $T_{call} = T_{checkpoint} + T_{rtt} + T_{proc}$. As above, $T_{rtt} + T_{proc}$ represents the time it takes to perform the actual call to the node. In addition, one has to pay the cost of a checkpoint operation $T_{checkpoint}$, which can be expressed as $T_{rec} + T_{transfer}$, where T_{rec} is the time it takes for the leader replica to record its state and $T_{transfer}$ is the time needed to transfer the checkpoint image to the follower replicas via reliable multicast. The state recording delay T_{rec} depends on the number and size of data objects that were created/modified by the mission program. The image transfer delay can be expressed as $T_{transfer} = ImageS/MaxP \times T_{rm}$, where $ImageS$ is the size of the image, $MaxP$ is the maximum packet payload for the underlying network and T_{rm} is (as above) the time it takes to send a single reliable multicast message that occupies a full network packet.

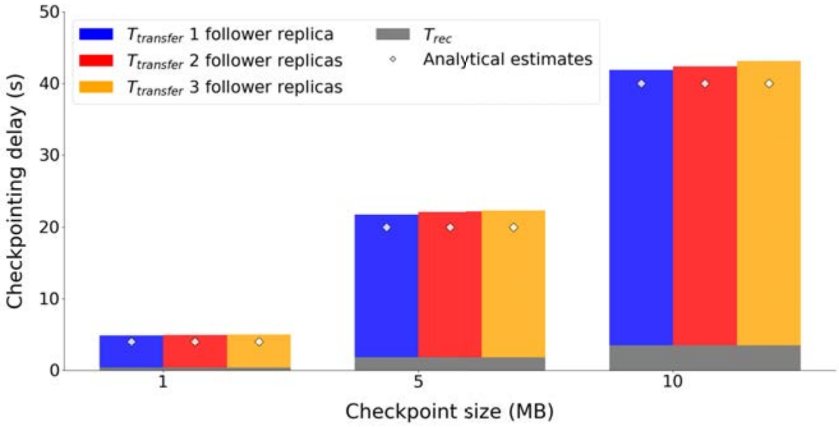


FIGURE 4.7: Checkpointing delay in non-deterministic execution

We have measured T_{call} for a null service, with $T_{proc} = 0$ and $T_{rtt} = 8$ milliseconds for an empty service call request and reply over the 1 Mbps WiFi network. We do this for 2, 3 and 4 replicas (1, 2 and 3 follower replicas respectively). Also, we artificially vary the size of checkpoint images, from 1 MB, 5 MB up to 10 MB, which are representative sizes for several test applications we have programmed in TeCoLa.

Figure 4.7 reports the results together with the analytical estimates. Given that the cost for performing the call itself is negligible compared to the checkpoint delay $T_{checkpoint}$, we only show the latter, broken down to the state recording delay T_{rec} and the image transfer delay $T_{transfer}$. As expected, the delay grows linearly to the size of the checkpoint image. Note that the number of replicas does not affect the checkpointing delay significantly; for every additional replica, the increase is lower than 1% thanks to the efficient underlying reliable multicast implementation (as discussed above, $T_{rm} = 6$ milliseconds). However, it is clear that taking a checkpoint at every service call incurs a significant penalty, especially when the execution state is large. It is thus important for the mission program to accurately indicate the parts that are non-deterministic, allowing the system to adopt the mode of deterministic execution as much as possible.

Chapter 5

Tolerance of Byzantine Failures

5.1 Contributions and Outline

In this chapter, we study how to tolerate Byzantine failures of the mission controller in centrally coordinated drone missions and propose an active replication scheme that relies on synchronous communication with signed messages and tolerates f Byzantine failures using $N = 2 \times f + 1$ replicas. We consider such a feature essential for applications where the mission should progress normally and correctly despite the Byzantine (i.e., arbitrary) behavior of the controller which may be due to malicious attacks or software errors.

While the fundamental principles of active replication for Byzantine fault tolerance have been laid out a long time ago, active replication is traditionally studied for the server / service side of computer systems [155]. On the contrary, in our case the most critical system component is the mission controller, which invokes the drones acting as their client, mandating a different approach. Furthermore, the mission dynamics due to possible failures of individual drones require special handling to ensure a consistent execution of the mission.

Our main contributions are the following:

- We describe, in detail, the approach and system-level mechanisms for handling a Byzantine behavior of the mission controller.
- We informally argue about the correctness of the proposed approach.
- We capture the overheads of a prototype implementation in a simulated environment.

The rest of the chapter is organized as follows. Section 5.2 presents the key assumptions regarding the operation of the system model. Section 5.3 describes the mechanisms that achieve the desired Byzantine fault-tolerant functionality. Section 5.4 discusses a prototype implementation of the proposed approach. Finally, Section 5.5 presents indicative performance results through simulation experiments and discusses the actual practicality of our approach.

5.2 System Model

In our system model, drones are modeled as *nodes* that expose the available sensing, actuation and mobility resources in the form of *services* that are being invoked remotely by a *mission program* that describes the high-level mission objectives. The same mission program is actively executed by multiple *mission controller replicas*, and each replica invokes the nodes inside a separate *replica-node communication domain*. Replicas communicate through the *replica domain*, whereas nodes do not need to interact with each other for the purposes of the mission execution. For more details, refer to Chapter 3.

The key assumptions are summarized as follows:

- Remote node service invocations have at-most-once semantics [105].
- Node failures follow the fail-stop model, i.e., a failed node stops operating.
- Node failures are detected by the replicas at the system's transport/RPC layer.
- Mission controller replicas can experience Byzantine failures, i.e., they may behave arbitrarily, appearing inconsistently both failed and functioning to different failure detection systems.
- Messages (at all communication domains) are signed, and all entities know each other's public encryption keys in order to verify signatures.
- Third parties are computationally bounded so they cannot forge signatures to conduct masquerading or injection attacks.
- There is a known upper bound for reliable message delivery in the replica-node domains and in the replica domain.
- The node's request processing time is known. In practice, it suffices to have a conservative upper bound for each type of service call.
- Mission programs are deterministic.
- Mission controller replicas execute the mission program at the same speed. This can be achieved by running the mission controller software on machines with the same processing capacity on top of a real-time operating system.

Regarding the upper bound delivery guarantees, modern wireless technologies like 5G can achieve high data rates in the gigabit scale, latency in the order of millisecond and ultra-high reliability [182, 126]; thus, we consider that such an assumption is realistic even for long-range communication.

5.3 Byzantine Fault Tolerance Mechanisms

At first, in section 5.3.1, we present the desired properties of a fault tolerance mechanism. Next, in section 5.3.2, we present an overview of the proposed approach that satisfies these properties. Finally, in the remaining sections we discuss our approach in more detail, in an incremental way.

5.3.1 Fault tolerance properties

The objective is to tolerate failures of the mission controller in a way that is fully transparent to the mission program. We capture the desired functionality in the form of the following properties:

- **Byzantine Fault Tolerance.** It is possible to tolerate not just simple fail-stop failures, but also Byzantine failures of the mission controller. In particular, a Byzantine replica may send a corrupt request to the node. It may also send a

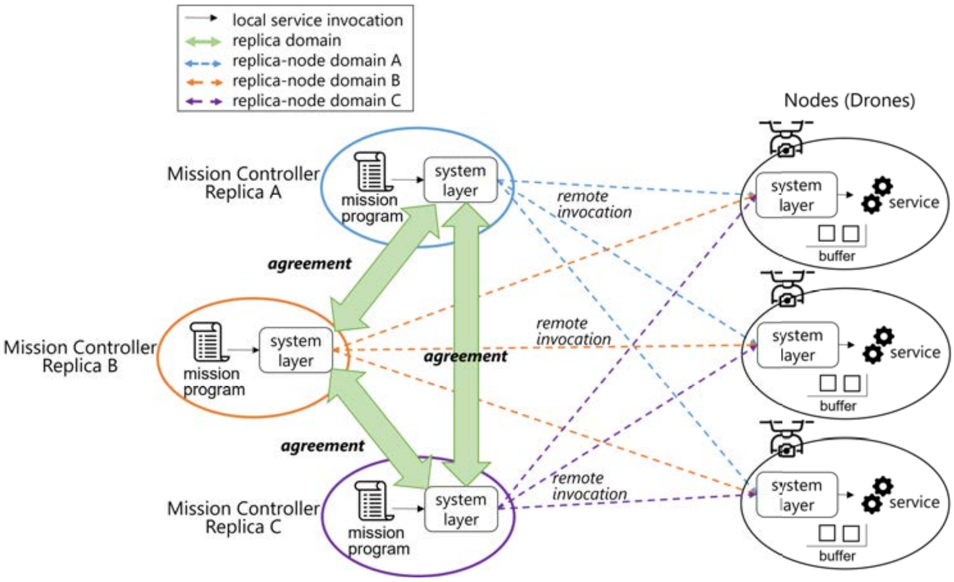


FIGURE 5.1: Overview of the system entities, their interactions and the introduced mechanisms

request much earlier/later than the correct replicas or perhaps it may not send any request at all.

- **At-most-once Invocation Semantics.** The at-most-once semantics of service invocation assumed by the mission program are preserved in a transparent way.
- **Synchrony and Consistency.** All correct replicas of the mission controller execute the node invocations of the mission program in the same order and in synchrony, despite the mission dynamics that can be caused by node failures.

These properties can effectively provide the illusion of a single "perfect" mission controller that never fails/malfunctions and always executes the mission program in a correct way.

5.3.2 Solution sketch

Our solution assumes $N = 2 \times f + 1$ replicas for tolerating up to f Byzantine failures. The main steps that are performed when the mission program invokes a node service are as follows:

1. The replicas send a service invocation request to the node.
2. The node considers and starts processing the request only if it has received identical copies from the majority of the replicas ($f + 1$).
3. The node executes the request and sends the same reply with the result to all replicas.
4. The replicas wait a maximum time to receive a reply from the node. If no reply is received within the given amount of time, the replicas infers that the node has failed silently.

5. The replicas, before proceeding with the execution of the mission program, run a Byzantine agreement protocol to decide the outcome of the node invocation that will be communicated to the mission program.

Figure 5.1 illustrates the entities, the introduced mechanisms and the interactions that take place in our system model. In the sequel, we start with the basic interaction that takes place between the replica and the node to perform a service invocation. We then discuss the case where there are no failures and all replicas of the mission controller function correctly, followed by the different cases of Byzantine behavior. Finally, we explain how to deal with the mission dynamics that can be caused by node failures.

5.3.3 Basic service invocation

Every service invocation performed by the mission program is converted into a blocking request-reply interaction between the respective mission controller replica and the target node. The mission controller sends a request to the node, which contains the replica identifier, a sequence number and the request parameters. The node stores incoming requests in a buffer until it receives the same request (same sequence number and same request parameters) from the majority of the replicas ($f + 1$). The node then processes the request and sends a reply with the result to all replicas.

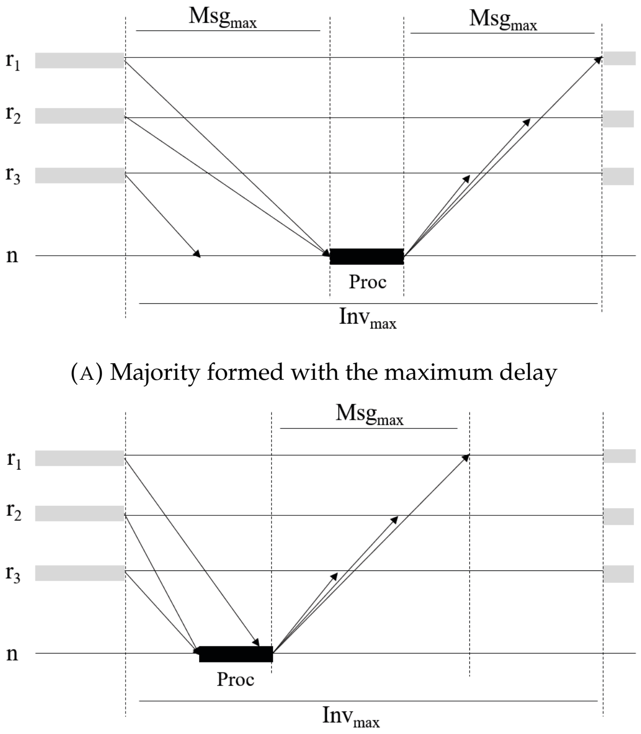
For the purpose of the following presentation, let Msg_{max} denote the largest possible message delivery delay in the replica-node domain. Also, let $Proc$ be the amount of time that is needed for a node to process a given service request.

5.3.4 Fault-free operation

Recall that the mission program is deterministic, and replicas execute the mission program at the same speed. Thus, they are expected to perform the same node invocations in the same order and at the same time.

Figure 5.2 illustrates two indicative scenarios for a configuration with three replicas that function correctly. In Figure 5.2a, the requests of r_1 and r_2 arrive at the node with the maximum delay Msg_{max}. Any of these two requests suffices to form the majority, combined with the request of r_3 that was received earlier. After the request is processed, the node sends the reply to all replicas. In this scenario, replica r_1 experiences the largest possible node invocation delay $Inv_{max} = 2 \times Msg_{max} + Proc$. In Figure 5.2b, the majority is formed at an earlier point in time, when the requests of r_2 and r_3 arrive at the node. Thus, processing starts before receiving the request of r_1 and the invocation completes with a smaller delay than in the previous scenario.

Note that in order for all replicas to resume the execution of the mission program at the same point in time, every replica has to wait for Inv_{max} from the point of invocation, even if it receives the node's reply earlier. In the scenario of Figure 5.2a, r_2 and r_3 have to wait in order to be in synchrony with r_1. In Figure 5.2b, all replicas have to wait. Although this waiting period is partly superfluous, since the node's reply arrives at all replicas earlier than in Figure 5.2a, the replicas have no global view and cannot distinguish between the two scenarios.



(A) Majority formed with the maximum delay

(B) Majority formed earlier

FIGURE 5.2: Node invocation with correct replicas

5.3.5 Corrupt requests

A replica that experiences a Byzantine failure may send a corrupt request to the node. Such requests are properly handled via the majority rule. As discussed above, the node does not process a request unless this was sent from the majority of the replicas ($f+1$). Thus, even if all f Byzantine replicas sent the same corrupt request, this will be ignored as it will not gather the necessary majority.

Figure 5.3 shows an indicative scenario with two correct replicas r_1 and r_3, and one Byzantine replica r_2 that sends a corrupt request. The request of r_2 arrives after the request of r_3 but before the request of r_1. The correct request is identified only when the request of r_1 arrives, at which point the node starts processing it. The arrival time of the corrupt request is irrelevant as this does not affect the formation of the majority. Also, the maximum invocation delay Inv_{max} remains the same as when all

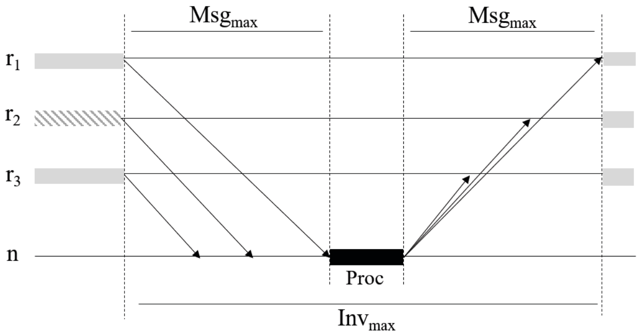
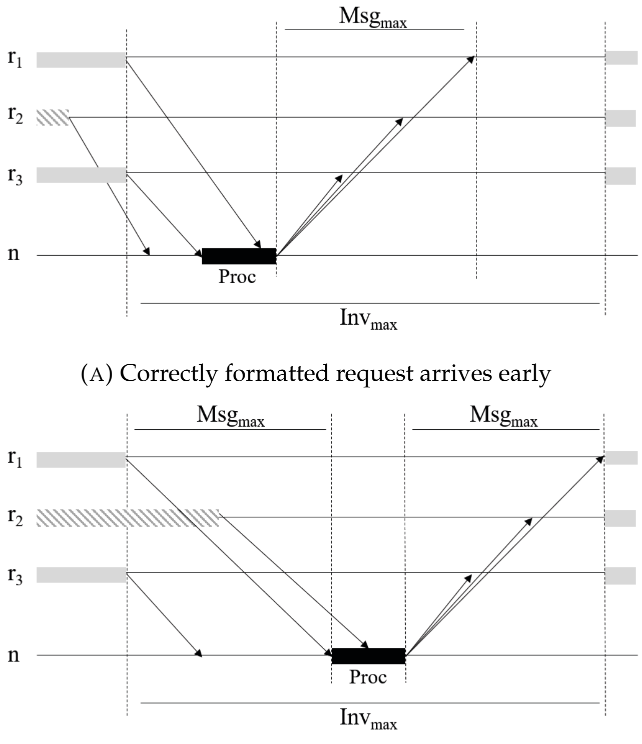


FIGURE 5.3: Node invocation with a corrupt request



(A) Correctly formatted request arrives early

(B) Majority formed earlier

FIGURE 5.4: Node invocation with ill-timed requests

replicas function correctly.

Importantly, Byzantine replicas cannot flood the node's buffer with arbitrary requests. Since each replica is expected to execute the next node invocation only after having completed the previous one, the node cannot have more than one outstanding request for each replica in its buffer (any other request that is sent by that replica can be simply dropped). Further, once the node identifies and processes a correct request, it can safely remove/drop all requests with a smaller or equal sequence number.

5.3.6 Out of sync and omitted requests

A Byzantine replica may send a correctly formatted request that is badly timed (out of sync), which means that it arrives at the node too early or too late compared to the correct replicas. Also, a Byzantine replica may experience an omission failure, which means that it may not send a request at all or that the transmission may fail. Figure 5.4 shows two indicative scenarios with two correct replicas r_1 and r_3, and one Byzantine replica r_2.

In Figure 5.4a, replica r_2 sends its request earlier than expected. In this case, the needed majority is formed as soon as the request of the first correct replica r_3 is received. Therefore, the node will start processing the request before receiving the request of the second correct replica r_1. This is not a problem at all, in fact it is effectively equivalent to the scenario of Figure 5.2b for fault-free operation. In general, the majority for a given request can be formed only after having received it from at least one correct replica, which, in turn, ensures that all other correct replicas have also sent the same request to the node and are expecting a reply.

Figure 5.4b shows a scenario where the Byzantine replica r_2 sends a request later than expected. In this case, the majority is formed based on the requests of the correct replicas r_1 and r_3. When the request of r_2 arrives, it is simply ignored/dropped. Again, this is equivalent to the scenario for fault-free operation that is shown in Figure 5.2a.

Note that Figure 5.4b also captures the scenario where the Byzantine replica does not send any request at all, due to an omission or transmission failure. In this case, the node will process the request exactly as discussed above.

5.3.7 Node failures

The assumption that all replicas execute the mission program in exactly the same way can be invalidated if a node fails during service invocation. This is because one replica may receive the node's reply, while another replica may not receive the reply and detect a node failure, leading to a different execution flow in the mission program at these replicas.

To ensure identical execution of the mission program, all replicas must return the same invocation result to the mission program, even if the target node experiences a failure during invocation. To this end, after the basic node invocation completes, the replicas run an agreement protocol to decide about the outcome of the invocation that will be reported to the mission program.

It suffices for the replicas to achieve Interactive Consistency [127], where each replica r_i votes for a value v_i. Correct replicas vote the outcome of the node invocation that they have actually experienced: the reply received from the node or a node failure if no reply was received within the bounds of the worst-case invocation delay Inv_{max}. Byzantine replicas may vote for an arbitrary value or refuse to send some of the expected messages during the agreement protocol. Given that messages are signed, a Byzantine replica cannot change the contents of a message that was sent by a correct replica. At the end, each (correct) replica decides for a vector d where element $d[i]$ corresponds to replica r_i. If this is a correct replica, $d[i]$ is equal to v_i, else $d[i]$ can take any value but this will be the same at all correct replicas. The decision about the outcome of the node invocation at each replica is v if at least $f + 1$ of the vector elements $d[i], 1 \leq i \leq N$ are equal to v, else the decided outcome is that the node has failed.

Recall that there are $N = 2 \times f + 1$ replicas. Therefore, if all correct replicas receive a reply from the node, the decided outcome for the node invocation will be that reply. If all correct replicas detect a node failure, the decided outcome will be a node failure. If the node fails during the invocation and some correct replicas receive a reply from it while some other correct replicas do not receive a reply (detect the failure), the decision depends on the vote of the Byzantine replicas. Notably, if the final decision is that the node has replied, it is certain that the agreed reply is the one that was actually sent by the node to at least one correct replica (else this value cannot have an $f + 1$ majority). If the final decision is that the node failed, it is certain that some correct replicas actually detected a node failure. In other words, the Byzantine replicas cannot change the reply that was sent by the node, nor can they introduce a phantom node failure, nor can they hide an actual node failure.

In any case, all (correct) replicas will decide on the same outcome. If the decision is that the invocation succeeded, all replicas will return to the mission program the

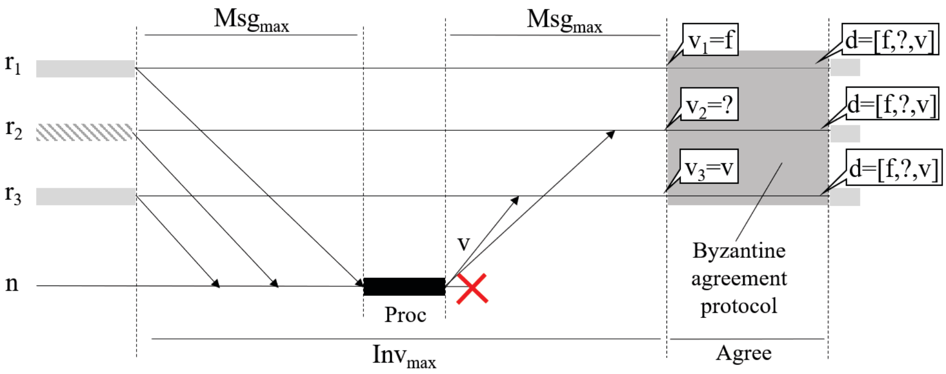


FIGURE 5.5: Invocation with a node failure

result that was included in the node's reply. If the decision is that the invocation failed, they will all return an error to the mission program indicating that the node failed. Thus, the mission program will follow the same execution path at all replicas. Note that the replicas may decide that the invocation failed even though the node may have managed to process the request and send a reply to some of the replicas. This is consistent with the at-most-once semantics of node invocations.

Figure 5.5 shows an indicative invocation scenario for the case of a node failure, with two correct replicas r_1, r_3 and one Byzantine replica r_2 that sends a corrupt request as in Figure 5.3. In this case, the node fails right after sending its reply v to one of the correct replicas r_3 and to the Byzantine replica r_2. The second correct replica r_1 does not receive a reply and detects the node failure. In the agreement protocol that follows, r_3 truthfully proposes $v_3 = v$ (the reply it received from the node) while r_1 also truthfully proposes $v_1 = f$ (the experienced node failure). The final decision depends on the proposal v_2 of the Byzantine replica r_2. If $v_2 = v$, the common decision at all replicas will be v, which is logically consistent since the node actually generated that reply. Else, if r_2 proposes any other value or remains silent and does not participate in the agreement protocol, replicas r_1 and r_3 will decide f (node failure), which is also consistent with the at-most-once semantics of service invocation.

To maintain synchrony, the replicas commence the agreement protocol Inv_{max} after the point of invocation. For f Byzantine failures, the agreement takes $f + 1$ communication rounds [37]. Assuming a message is delivered to all replicas latest within $RMsg_{max}$ and that the transmissions that are performed concurrently by the N replicas in each round are serialized over the underlying network, agreement is reached in $(f + 1) \times N \times RMsg_{max}$. If, however, the replica domain allows concurrent transmissions to be performed in parallel, the agreement time is $(f + 1) \times RMsg_{max}$. The replicas agreement time is added to Inv_{max} to give the total node invocation delay experienced by the mission program.

5.4 Implementation

We have developed a proof-of-concept implementation of the proposed replication approach in the TeCoLa programming framework [85] that supports the high-level

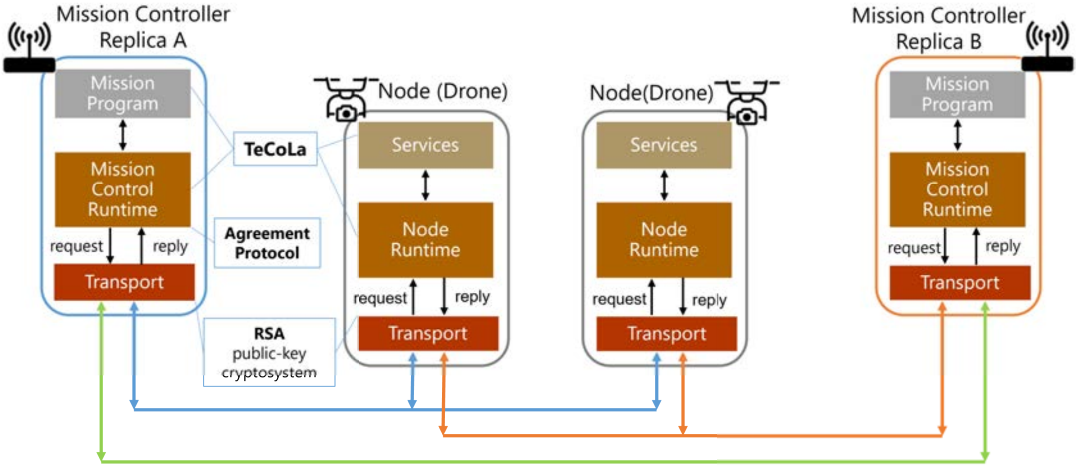


FIGURE 5.6: Software architecture of the prototype

coordination of teams of unmanned vehicles like drones. The software organization of the mission controller and node entities is shown in Figure 5.6. In our experiments, we employ several instances of the mission controller that execute the mission program in parallel.

The interaction between the mission program and the node services is implemented through suitable middleware and runtime support at both the nodes and the mission controller. The runtime environment of the mission controller is responsible for the execution of the mission program, while the node runtime environment supports the service-oriented access of the node's mobility and sensing/actuation capabilities. The request-reply interaction between the replica and the node is implemented in the spirit of remote procedure calls (RPCs). The number of transmission attempts can be flexibly configured, allowing this to be set according to the reliability of the underlying communication channel.

The node runtime environment handles incoming requests as discussed in the previous section. When a correct request is identified, the local service is up-called and the result is returned as a reply. In addition to the request-reply protocol for the service invocation, the mission controller runtime environment implements the agreement protocol between the replicas. In each communication round, every replica sends to all other replicas a message with the required information: initially its own vote and in subsequent rounds the votes it received from the other replicas. As an optimization, in each of the $f+1$ communication rounds, each replica packs all its messages in a single multicast message that is sent to all other replicas using UDP/IP multicast (assuming an underlying network with native multicast capability). When a replica receives such a multicast message, it unpacks it and uses only the information relevant for it. Thus, the communication cost for the agreement protocol is $(f+1) \times N$ multicast messages. Again, the number of transmission attempts is configurable.

For message authentication, we use the RSA public-key cryptosystem [148] to generate digital signatures that provide message authentication, message integrity and non-repudiation. The messages are signed before transmission and verified upon reception. All mission controller replicas and nodes have each other's public keys pre-installed locally in order to perform the verification process without requiring the transmission of the public key along the messages.

5.5 Evaluation

In this section, we report and discuss the overhead of service invocation, which was experimentally measured using a controlled simulation setup.

5.5.1 Experimental setup

Our measurements are performed using the simulation environment we have developed and presented in chapter 2, section 2.3, where different virtual drones (v-drones) can be controlled from one or more virtual ground stations (v-GSs). The v-drones run the node software stack and the v-GSs run the mission controller software stack. Each of these entities is packaged as a Linux-based Virtual Machine (VM) with 4 vCPUs and 3GB of RAM on top of the KVM hypervisor [97] running in a cloud-like computing infrastructure. The clocks of the guest VMs are synchronized to the clock of the host machine with a sub-microsecond accuracy, using the virtual PTP hardware clock (PHC) mechanism of KVM. The mission controller replicas start the execution of the mission program synchronously, at a given time that is specified via a configuration parameter.

Each replica-node domain is mapped to a separate WiFi network used for the communication between the specific replica and the v-drones. The WiFi networks are simulated using ns-3 [147] configured to the 802.11b standard. The replica domain used for the communication between the v-GSs is implemented via the default KVM networking facility as a shared Ethernet network. Replicas send messages to each other using the native multicast capability of the network. Concurrent message transmissions are serialized.

To capture the raw overhead of the fault tolerance mechanism for typical coordination commands (these do not include large payloads), we use a mission program that performs a series of null service calls to a drone node. We use a configuration with $N = 3$ replicas of the mission controller that can tolerate $f = 1$ Byzantine failure; we expect this reliability level to be sufficient for most applications.

5.5.2 Basic costs

Based on the time it takes to transmit a null service request-reply message in the replica-node domains, we set Msg_{max} to 3 ms for a maximum number of three transmissions in the RPC transport layer. On top of this comes the message packing and digital signature overhead at the sender and the unpacking and signature verification overhead at the receiver, for which we allow another 1.5 ms. Null requests are processed instantly, so $Proc$ is assumed to be zero.

To tolerate $f = 1$ Byzantine failure, the agreement protocol between the mission controller replicas involves $f + 1 = 2$ communication rounds. We set $RMsg_{max}$ to 11.5 ms, based on the transmission time for multicast messages in the replica domain, for a maximum of three transmissions. Also, in each communication round, we allow 5.5 ms for the packing/signature of the messages sent and for the unpacking/verification of the messages that are received by the replicas.

Major cost item	**time (ms)**	**%**
Request-reply processing	3.0	3.5%
Request-reply transmission	5.5	6.5%
Vote processing	10	11.8%
Vote transmission	66.5	78.2%
Total	85.0	

TABLE 5.1: Actual costs of node invocation (worst-case)

5.5.3 Node invocation overhead

We measure the node invocation delay for over 100 consecutive invocations performed by the test mission program, with all replicas of the mission controller working properly. Note that failures do not affect system operation since the Byzantine fault tolerance mechanism works based on the worst-case timing bounds.

The average delay that is experienced by the mission program, with the Byzantine fault tolerance mechanism configured to use the above timings, is roughly 90 ms. Table 5.1 gives a breakdown of the actual costs for the case where the full number of transmissions is performed to achieve successful message delivery in both the replica-node and replica domains. The message processing costs include the respective packing/signature and unpacking/verification overheads. It can be inferred that the internal timing of the Byzantine fault tolerance mechanism is not too conservative given that the invocation delay at the level of the mission program is close to the total actual costs.

We also observe that 90% of the cost is due to the overhead of the agreement protocol between the replicas, which is clearly the performance bottleneck. On the one hand, this speaks in favor of using a faster network for the replica domain and/or cheaper message signature and verification methods. For instance, if the replicas would communicate over a fast Gigabit Ethernet, this could reduce the vote transmission time to 1/4. Based on Table 5.1, this would, in turn, reduce the actual invocation delay to about 36 ms, a substantial 2.4x improvement. On the other hand, a slower communication between the replicas and the nodes would have a relatively small impact. As an example, if the communication delay in the replica-node domain increases by 3x, the estimated node invocation costs would be around 96 ms, just 1.13x higher than the current value.

5.5.4 Results discussion

Recall that the mission controller is responsible only for the high-level coordination of the drones. In particular, it is *not* involved in tight control loops that concern the stability of the vehicle and obstacle avoidance and is not expected to interact with the nodes very frequently. Therefore, invocation delays even in the order of 100 ms are quite acceptable. To give a concrete example, in a mission where the drones move at a speed of 10 meters/second, the above invocation delay roughly translates to a distance of 1 meter, which is fine for high-level coordination purposes. Note that if certain actions require high position accuracy, the mission program can explicitly wait for the drones to reach the required position, before invoking the service that performs the action.

Tolerance to a single Byzantine failure at such high overhead may not seem attractive, at first. Nevertheless, in practice, this is already a very significant capability for this particular type of system that can make all the difference between success and disaster. Also, since Byzantine failures are expected to be rare, it is unlikely that a higher degree of Byzantine fault tolerance needs to be provided for the mission control system. Even if this should be required, acceptable performance could still be achieved using a fast network setup for the replica domain.

Chapter 6

Related Work

In this chapter, we discuss related work in the field of fault-tolerant systems and the differentiation between our research and previous efforts. We organize literature discussion across three sections: the toleration of fail-stop failures, the Byzantine fault tolerance, and works specific to autonomous/robotic systems.

6.1 Tolerance of Fail-Stop Failures

Extensive research has been done in regard to fault tolerance in distributed systems. Among the most well studied techniques are rollback recovery [41] and replication [53].

Rollback protocols assume a stable storage that is used to store recovery information during normal execution. This state recording operation can be done in an independent/uncoordinated [15], coordinated [25] or communication-induced way [150]. After a failure occurs, the failed component uses this information to restart its execution from a more recent state. To reduce the extent of rollback and guarantee that the pre-failure execution can be deterministically regenerated, log-based protocols [3] have been proposed. Such log-based methods enable a system to recover beyond the most recent set of consistent checkpoints by combining checkpointing with logging and replaying of non-deterministic events, such as messages/interaction with other systems. Thus, they are particularly attractive for applications that frequently interact with the outside world.

Software-based replication can be characterized as passive or active and its main purpose is to achieve fault tolerance of critical components by employing multiple instances of them that can fail independently. In the passive replication technique [20], also known as the primary-backup approach, one of the replicas, called the primary, is responsible for receiving and processing input and producing output. The remaining, called the backups, are merely notified to apply the changes produced by that processing. On the contrary, active replication [155], also known as state machine approach, is a non-centralized approach where all the replicas receive and process (concurrently) the same sequence of inputs. This leads to masking the failures and achieving better performance, making it ideal for (soft) real-time applications. However, it requires that the output produced by all the replicas is the same, i.e., the processing is deterministic. Viewstamped Replication [123] and Paxos [98] are two such well-known protocols, designed to tolerate f fail-stop failures using $N \geq 2 \times f + 1$ replicas. In addition, there are variations that combine elements from both passive and active replication strategies [176].

There are also fault tolerance schemes that combine rollback recovery with replication. For instance, [142] proposes a mechanism to reduce the expected completion time of long running computations in the presence of failures. To achieve this, it uses multiple active replicas that take checkpoint in case another replica fails. Then, this checkpoint state is used to restart the failed replica.

While the vast majority of the replication-related bibliography focuses on replicating the server side, in our work we apply replication to the client side (the mission controller), which is the most critical component. Also, traditional active replication requires coordination between the replicas during each request in order for them to be handled in the same order, whereas in our approach the replicas need to synchronize only when a node fails. Finally, to support non-deterministic applications, we adapt our approach to a semi-active technique supported by appropriate checkpointing.

6.2 Byzantine Fault Tolerance

Byzantine fault tolerance (BFT) is the dependability of fault-tolerant computer systems to random/spurious hardware or software faults, or malicious attacks. Active replication (or state machine replication) is the general technique used to support such functionality. In this case all replicas receive and process in parallel the same sequence of inputs, and the individual outputs are then combined to determine the final result, typically following a majority scheme to deal with Byzantine behavior.

There are many research efforts on making Byzantine fault-tolerant systems practical and efficient. Practical Byzantine Fault Tolerance (PBFT) [24] tolerates f Byzantine failures with $N \geq 3 \times f + 1$. The safety property is preserved even under asynchrony, but progress is made only during synchronous periods. Zyzzyva [79] improves performance through systematic exploitation of speculation in the replicas, which automatically adopt the order proposed by the primary and use agreement protocols only in case divergence is observed by the clients. [10] improves the robustness by executing multiple instances of the Byzantine fault tolerance protocol, each with the primary replica placed in a different physical machine. [74] provides better resource efficiency, by using fewer active replicas ($f + 1$) for both agreement and execution during normal operation, while in case of fault detection or suspicion it switches to a more resilient agreement protocol. The authors in [174] present a trusted/tamper-proof service for assigning unique and monotonically increasing identifiers to messages. Based on this, they present improved versions of PBFT and Zyzzyva that achieve Byzantine fault tolerance in asynchronous systems for only $2 \times f + 1$ replicas with the minimum number of communication steps.

All the above work concerns the fault tolerance of server-side logic towards its clients. In contrast, in the system model for coordinated missions that we consider in this thesis, our work addresses the problem of a potentially faulty client that sends commands to different independent servers.

The problem of consensus is fundamental to distributed systems research. The FLP impossibility result [47] proved that in a fully asynchronous system there is no deterministic algorithm that can achieve consensus in the presence of even just one fail-stop failure. In synchronous systems, the Byzantine agreement problem [100] and the equivalent interactive consistency problem [127] are solvable with $N \geq 3 \times f + 1$ processes, without using authentication (signed messages), but have exponential message complexity. In settings where there is authentication support so that processes can safely sign their messages (and proposed values), the above agreement problems can be solved with just $N \geq f$ processes.

Fully polynomial protocols for both the authenticated [37] and the unauthenticated [51] settings require $f + 1$ communication rounds. To circumvent this lower bound, the use of randomization has been explored in [141], leading to expected constant-round protocols for both the authenticated [75] and the unauthenticated [45] settings. Such algorithms could be employed for the synchronization between the replicas of the mission controller. But it would not be possible to give a strict upper bound for the node invocation delay, which can be important for the type of system we study in our work.

6.3 Fault Tolerance in Robotic Systems

Fault tolerance has also been studied in the context of robotic systems. In [78], a checkpointing and recovery protocol is presented for achieving resiliency against sensor failures / attacks in CPS systems. In addition, in [33], a framework is presented for the detection of both software and hardware failures, and the fault mitigation through self-adaption or cooperation between multiple robots.

In swarm robotics, tolerance to faults of single robots is essentially a built-in feature. Failures are typically detected through biologically inspired techniques [27, 76], which exploit the natural redundancy of the (large) swarm. Since robots do not have predefined roles, reorganization is achieved in a self-healing fashion [16, 166]. Still, even swarm-based systems may need to be driven by external high-level commands, which leads to the problem we study in this work.

There are also works targeting multi-robot collaborative systems, where multiple independent robots cooperate to accomplish a common goal/task. ALLIANCE [125] is a software architecture that facilitates cooperative control of teams of mobile robots for achieving fault tolerance. It is a distributed, behavior-based architecture that allows each robot to adapt its actions during a mission. This way, if a robot fails, its tasks are dynamically re-allocated to the remaining team members. [13] presents a programming abstraction for handling failures in ensembles of robots. The proposed abstraction allows the application programmer to annotate code blocks that include critical actions and define compensating actions in case a failure occurs. However, in both approaches, the fault handling and recovery is the responsibility of the developer, whereas our work practically achieves full transparency for the developer of the mission program.

The specific system model for centrally coordinated multi-drone missions that we consider in this thesis, comes with its own peculiarities, dictating a different approach. In [84], a passive replication approach is described, using a combination of checkpointing and logging to support a rollback and replay of the mission program in case the primary mission controller fails. However, it covers only deterministic mission programs and fail-stop failures. Differently, in this thesis, in order to support fail-stop failures of the mission controller, we propose an active replication approach in conjunction with a logging mechanism which can also utilize a semi-active replication scheme in order to support non-deterministic operations. In addition, we use active replication with an agreement protocol to address Byzantine failures under certain assumptions, which offers even more dependability to this kind of systems.

Part II

Managed Operation, Testing and Integration of Drone Applications

Chapter 7

Flexible Deployment and Safe Operation of Drone Applications

7.1 Contributions and Outline

In this chapter, we present a holistic approach towards supporting a more reliable managed operation of drone applications on a shared drone infrastructure. This is achieved through a software platform that takes care of the automated deployment and controlled execution of drone applications, coupled with corresponding simulation and digital twin support for detecting bugs before deployment and indicating possible malfunctions during operation in the real world, respectively. We deem that such an approach can offer various benefits. On the one hand, the software platform can support in a flexible way the execution of application code directly on the drone, making it possible to support data-driven applications with tight-control loops. At the same time, it can capture actions that violate certain restrictions posed by the authorities and handle them appropriately, ensuring that the application cannot behave in an arbitrary way. On the other hand, the simulation and digital twin support adds an extra safety layer to drone operations, contributes to building trust between the various stakeholders in such a drone applications ecosystem and could make drones more acceptable to the wider public.

Our main contributions are the following:

- We present the concept of a modular architecture combining a platform as a service (PaaS) system for drone applications, which offers automated deployment and restriction enforcement, with corresponding simulation and digital twin support that can be used to detect bugs before deployment and to indicate possible malfunctions during operation in the real world, respectively.
- We introduce suitable descriptors that capture and enforce the desired application functionality.
- We present in detail the design and the various configurations of the testing infrastructure.
- We discuss the most important aspects of a proof-of-concept implementation regarding both the PaaS system and the simulation and digital twin support.
- We showcase how the proposed work can be used in practice through representative case studies.

The rest of the chapter is organized as follows. Section 7.2 presents the overall system concept. Section 7.3 presents the design of the PaaS system and discusses the various metadata descriptions needed to support the desired application deployment and enforcement of restrictions. Section 7.4 presents the design of the simulation environment and digital twin support that we propose for this PaaS system. Section 7.5 describes the implementation of our proof-of-concept system, focusing on the components of the PaaS that provide the desired deployment and controlled execution functionality and the main aspects of the simulation and digital twin setups. Finally, Section 7.6 presents the experiments performed to validate our prototype using both the simulation and digital twin functionality for indicative test applications under different execution scenarios.

7.2 Concept

Several drone applications must run directly on the drone to perform certain operations on board. For instance, some applications need to control the navigation/ movement of the drone, including takeoff, path planning and landing. In addition, various sensing, processing and actuation tasks may have to be performed on the drone to support application-level control loops with minimal latency, or simply to avoid sending large amounts of raw data to the cloud. Such applications are currently deployed on privately owned drones in a manual way, while the responsible authorities that grant the flight permission do not have actual control on the operations that will be performed by the application at runtime.

Taking a different approach, we envision a system that provides drones as resources that can be used in a flexible way to run different applications directly on them in the spirit of fog/edge computing [18]. This should be done in a transparent and isolated way, so that the user does not have to deal with the management of the physical infrastructure or worry about side-effects due to application bugs.

In addition, the coexistence of multiple drones in residential areas, as it is being envisioned in the smart city concept, raises several safety and privacy issues and naturally leads to skepticism, limited acceptance by the public and, in rare cases, even to aggressive reactions [120]. To gain the citizen's trust, such systems have to be engineered to address safety and privacy by design, through suitable mechanisms that can be easily integrated and used to detect bugs and malfunctions.

In the following, we present the concept of such a system in detail.

7.2.1 Objectives

Our high-level goal is to enable a safer integration of drone applications in the urban environment, through suitable system support. The key objectives of the envisioned system are described below.

Simple and flexible application deployment. It should be possible to deploy applications on drones in a transparent way, in the spirit this is done today with cloud-based applications. This calls for a suitable application execution environment on the drone that provides isolation, along with minimal overhead and portability across different platforms. Furthermore, to automate the assignment of drones to applications, appropriate descriptive specifications are needed to capture the flight capabilities and sensing/computing/actuation resources of the available drones as well as the respective application requirements.

Strict compliance to safety and privacy restrictions. It should be possible to enforce safety and privacy constraints regarding drone operation. This calls for a central management system, which is aware of all the restrictions and the currently active applications and is responsible for giving the initial clearance to applications, combined with suitable monitoring mechanisms on the drone, which detect and handle violations at runtime. Furthermore, the potentially complex nature of these restrictions necessitates the explicit description of the application's expected behavior as well as an agreement between the application user and the management system regarding the actions that will be taken in case of (attempted) violations.

Integrated offline and runtime testing support. It should be possible for application developers to test both the platform and the application before deployment. This calls for a suitable simulation environment that can make it possible to perform a wide range of tests regarding both the platform itself and the applications that run on top of it. Furthermore, it is important to detect unexpected deviations and potential malfunctions of the drone at runtime. This can be addressed through appropriate digital twin support for the drone, that makes it possible to detect, at runtime, large deviations of the application execution from the expected behavior.

7.2.2 Main entities and stakeholders

We envision an open environment where (in principle) anyone can develop an application or provide a drone that can be used to run applications, and users are free to run any application at any point in time. The main entities, stakeholders and their interrelationships are shown in Figure 7.1.

Drone-based applications come in the form of independently deployable and runnable software components with certain sensing/actuation requirements as well as computing and communication requirements. The application may also require certain flight-related capabilities, such as hovering. In turn, drones serve as application hosts with specific sensor, actuator, computing and communication resources as well as specific flight/maneuvering capabilities, depending on the underlying hardware and flight platform, respectively.

The application developer implements a drone application, packaged so that it can run on top of a standardized runtime environment (the system may provide different options for this). The developer also specifies the drone resources and capabilities that are required by the application to function properly. Before an application can be registered with the system, it may go through a formal approval process, which may include code review and extensive testing in simulators or real-world testbeds.

The drone provider owns a drone to be made available for third-party applications. In order for a drone to be registered with the system, it has to pass flight-readiness tests and be covered by proper insurance. The provider also specifies the drone's resources, capabilities and limitations in terms of sensing, actuation, computing, networking/communication and flight ability. Lastly, one of the system's approved application runtime environments has to be installed on the drone.

The management authority operates the drone management system. It also provides the certified application runtime environment(s) for the drone platforms along with the various components that constitute the testing infrastructure. It also defines the restrictions regarding the usage of the airspace, like flight altitude, no-fly zones as

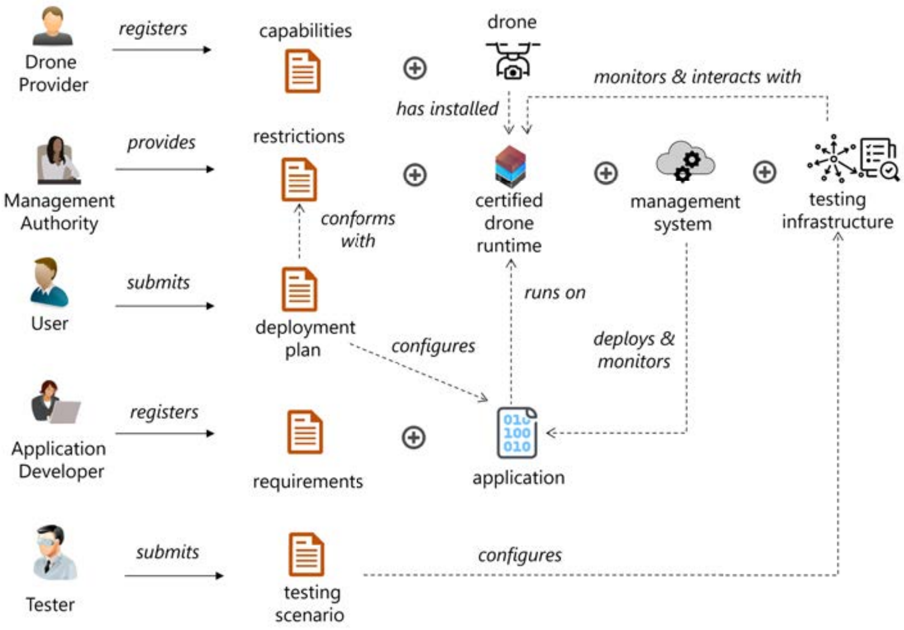


FIGURE 7.1: Main entities, stakeholders and relationships

well as areas where it is not allowed to use specific sensors or actuators of the drone platform. These restrictions create the regulatory context in which applications have to operate. Some of them may stem from standard flight safety regulations, while others may have to do with the geographical morphology or the intended use of specific areas. Also, it provides the testing infrastructure consisting of a suitable simulation facility.

The user selects a registered application and submits a corresponding deployment plan for execution. The area of operation is declared in the plan (this can be a set of waypoints or an entire area, depending on the application) and the user may select one of the registered drones to run the application, or let the system pick one that satisfies the application's requirements. As a last step, the user has to agree on any restrictions that apply to the area of operation and on the specific actions that will be taken by the system in case of violation. Once approved, the deployment plan serves as a contract regarding the behavior of the application.

Finally, the tester can be anyone from the previous stakeholders. It composes and submits a testing scenario consisting of the description of the required setup and the checks that should be done during execution and after completion. The scenario configures accordingly the testing infrastructure, which may include setting up a simulation environment and/or interacting with real drone(s). The results are collected to a central repository and, depending on the nature of the test, they are displayed to the tester during the scenario execution or after its completion.

7.3 PaaS Approach

In section 7.3.1, we present the high-level design of the platform as a service system, and in section 7.3.2 we discuss the structured descriptions that drive the automated application deployment, monitoring and violation handling.

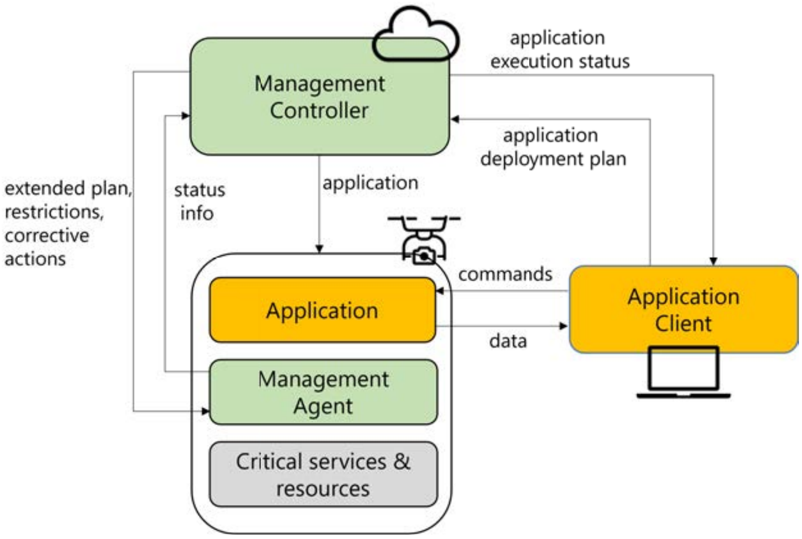

Management Controller
application execution status
application deployment plan
extended plan, restrictions, corrective actions
status info
application
commands
data
Application
Application Client
Management Agent
Critical services & resources

FIGURE 7.2: Main system components and basic interactions

7.3.1 Overview

Figure 7.2 presents the high-level system architecture, while Figure 7.3 illustrates the basic application deployment and monitoring process. The role of each of the system component is briefly discussed in the sequel.

The *management controller* runs in the cloud and keeps track of all drone-based applications that are currently running or are scheduled to run. The controller also determines whether a newly submitted application plan can be accepted for execution. More specifically, to approve a plan, the controller checks its requirements vs the available resources, other application plans and the various restrictions that may be relevant for the application at hand. If there are no conflicts, the application is deployed and starts running on the selected drone.

The *management agent* runs on the drone and is responsible for monitoring the operation of the application, detecting invalid behavior and enforcing corrective actions. As part of the application deployment process, the agent receives from the controller information about the intended / agreed area of operation as defined in the deployment plan, the restrictions that may apply to that area and the respective actions to be taken. During application execution, the agent intercepts through a monitoring module all application requests towards the sensors and autopilot of the drone, to check that these conform with the restrictions. In case of a violation, the request can be rejected and/or some corrective action may be applied. If there is a major (or repetitive) violation, the agent takes over control and applies the predefined fail-safe action, e.g., landing at a safe location.

The management controller and the agent regularly communicate during application execution. The agent periodically sends status information updates to the controller, regarding the progress of the application, the current state of the drone, any reported violations and the corrective actions that were taken in response. On the other hand, the controller sends updates regarding the general or application-specific restrictions.

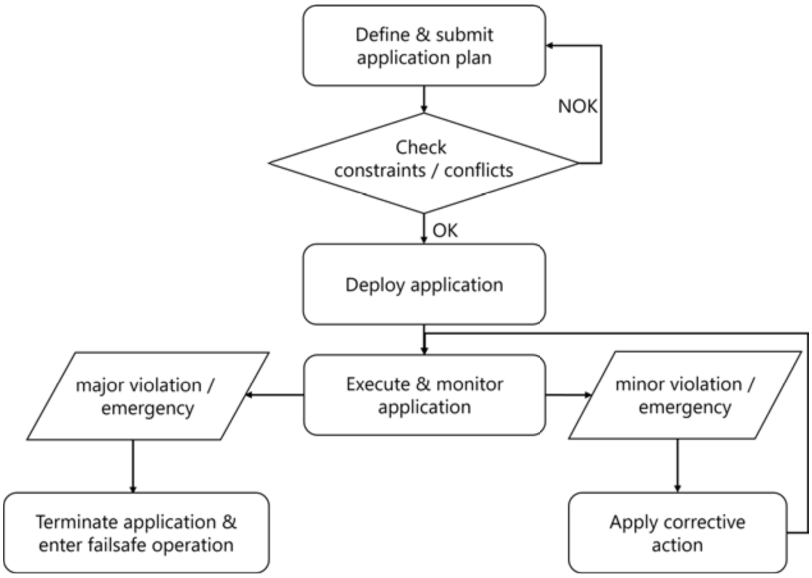

Define & submit application plan
NOK
Check constraints / conflicts
OK
Deploy application
major violation / emergency
Execute & monitor application
minor violation / emergency
Terminate application & enter failsafe operation
Apply corrective action

FIGURE 7.3: Application deployment and monitoring process

The management controller exposes a remote API that enables the submission of application deployment plans as well as the reception of updates regarding the deployment and execution status of applications. This API can be used to build interactive user interfaces and application-specific clients that may also communicate directly with the deployed application on the drone without going through the controller, as shown in Figure 7.2.

7.3.2 Structured descriptions

To enable the automated application deployment, monitoring and violation handling, several pieces of information must be described in an explicit way so that they can drive the corresponding mechanisms of the drone management system. More specifically, one needs to capture: (i) the resources and capabilities of each drone; (ii) the requirements and parameters of each application; (iii) the flight-related and sensor-related restrictions for the target environment (e.g., an urban area), along with the actions that can be taken in case of violations; (iv) the parameters of the submitted application deployment plans, including the approved flight plan, the relevant restrictions and corrective actions.

Below we provide indicative examples for these structured descriptions. For purposes of illustration, these are kept simple and focus on aerial vehicles with vertical take-off and landing capability (the most common type of drones used in urban environments). The actual descriptions are more elaborate and verbose; they are defined in JSON and YAML formats and are described by suitable JSON schemas that are also used to perform an automated validation.

Drone descriptions include general properties, physical and flight-related features and computing/sensing resources. An indicative example is given in Listing 1 for a custom quadcopter.

Application descriptors include the flight, computing and sensor requirements that need to be satisfied in order for the application to function in a satisfactory way. Part of the description includes the type of navigation configuration information the application needs in order to control the drone's movement as desired. Currently, there

Listing 1 Drone descriptor

```
id: 123
model: custom
type: quadcopter
category: small
# physical features
dimensions: { height: 30cm, length: 50cm, width: 50cm }
weight: 1200g
# flight features
autopilot: ArduCopterV3.6.11
max-speed: 15m/s
max-alt: 50m
max-time: 15min
capabilities: [hover]
# computing & communication resources
platform: RaspberryPi3
cpu: ARMv7l
ram: 1GB
storage: 5GB
os: Raspbian
networking: [WiFi, 4G]
# sensor resources
camera: { type: RGB, res: 1920x1080, model: ModelX }
```

are two such options: path-based and area-based. The former is for applications designed to move along a path defined as a list of waypoints that will be visited in sequence. The latter is for the case where the application will move within an area, but the exact path that will be followed within that area is not known in advance and is decided/adapted by the application at runtime. Listing 2 shows a descriptor for a surveillance application that expects as input a sequence of waypoints.

Listing 2 Application descriptor

```
id: 456
class: surveillance
navigation-type: waypoint-based
# flight requirements
max-speed: 10m/s
min-alt: 20m
max-alt: 30m
capabilities: [hover]
# computing & communication requirements
cpu: ARMv7l
ram: 512MB
storage: 1GB
os: Raspbian
networking: [4G]
# sensor requirements
camera: { type: RGB, res: 1280x720 }
# configuration
app-config: waypoints
```

The restriction descriptors capture the limitations posed by the management authorities regarding drone-based operations, including the respective corrective actions in case of an attempted violation. Each restriction concerns a specific area and can remain active continuously (until removed) or for a specific time interval. The target area is defined by a shape that can be two-dimensional, in which case the restriction is applied at all altitudes, or geometric, in which case the exact air volume is described. Also, a restriction can include the specific features of the drones and/or applications to which it applies. Listing 3 shows the descriptions of two restrictions: *R1* a temporary flight-related restriction (no-fly zone) that applies to small drones that execute surveillance applications, and *R2* a permanent sensor-related restriction (for the drone's camera) that applies to all drones and applications.

Listing 3 Restrictions descriptor

```
- id: R1
  class: flight-restriction
  type: no-fly-zone
  area: polygon{...}
  activate: 10 Feb 2019, 17:00 CET, 3hrs
  drone-features:
    category: small
    weight-less: 1500
  app-features:
    class: surveillance
  actions:
    - action: abort
      parameters: land-wp
    - action: correct-path
      parameters: target-point
- id: R2
  class: sensor-restriction
  type: no-photo-zone
  sensor: camera
  area: circular{...}
  activate: always
  drone-features: all
  app-features: all
  actions:
    - action: suppress
      parameters: None
```

Each restriction is associated with one or more repair action(s) that can be performed by the system in case of a violation. When multiple options are available, the user specifies the preferred one as part of the application deployment plan. For example, restriction *R2* in the above description has a single repair action, which is to suppress the usage of the camera. For *R1*, if the drone tries to move in the specified no-fly zone, one option is to abort the application and land the drone at a specific location. A more relaxed repair action is to correct the drone's course so that it remains within the approved flight path/area. In this case, the application is suspended until the correction is completed.

The different parameters of an application deployment plan are also captured via a corresponding descriptor, as shown in Listing 4. Some parts are provided by the user, like the application identifier, the target area/path and (optionally) the drone to be used to run the application. Other parts are generated by the system based on the features of the selected application and drone, like the assigned priority used for future deconflictions, the navigation radius used to set the hard boundaries of the air space that will be reserved for the application, and the restrictions that apply to the area/path of operation.

Depending on the type of restrictions that are included in the plan, the user must select the corrective action to be performed, if there are multiple options, and provide the necessary parameters, if any. In Listing 4, this information is in the *actions* configuration section. For restriction *R1*, we show indicative configuration settings for both options (the user has to choose one of them). More specifically, in case of the

Listing 4 Deployment plan descriptor

```
id: 789
# configuration
drone-id: 123
app-id: 456
activate: 10 Feb 2020, 17.00 CET, 10min
# app configuration
waypoints: [...]
# system generated
priority: normal
navigation-radius: 10m
restrictions: [R1, R2]
# user specified
actions-config:
  - restriction: R1
    action: abort
    land-wp: [39.3618020, 22.9336374]
  - restriction: R1
    action: correct-path
    target-point: intersection(issued-wp, R1-area)
  - restriction: comm-lost
    action: abort
    land-wp: [39.3618020, 22.9336374]
```

abort action the land waypoint is specified, whereas for the correct-path action the target location is specified as the closest intersection point of the waypoint issued by the application and the polygon area specifying the borders of the no-fly zone. Note that there is no entry for restriction *R2* as this has a single repair action that does not take any parameters. Finally, there is always a mandatory safety-related action, which is activated when the communication between the management system and the drone is disrupted. In this case, the default action is to abort the application and land the drone at the location specified in the plan (see last line of the description).

7.4 Simulation and Digital Twin Approach

In this section, at first, we outline our approach regarding the simulation and digital twin support. Next, in the remaining sections, we describe in detail the various testing configurations.

7.4.1 Overview

A high-level view of the testing framework is shown in Figure 7.4. The basic simulation entity is the so-called *virtual drone* (v-drone). It represents a simulated drone that can run the complete software stack of a real drone, including the autopilot and the Drone Runtime of the PaaS platform. The v-drone can be configured to work in two different modes: pure simulation mode and digital twin mode. Also, there is a *virtual Controller* entity (v-Controller), which is a mockup implementation of the real Controller of the PaaS platform, through which it is possible to issue commands to and receive status updates from v-drones. Simulation scenarios may involve multiple v-drones, which can communicate with the v-Controller and/or with each other through one or more simulated wireless channels. Depending on the mode of operation, virtual entities (v-drone and v-Controller) may co-exist and run in tandem with real system entities (drone and Controller).

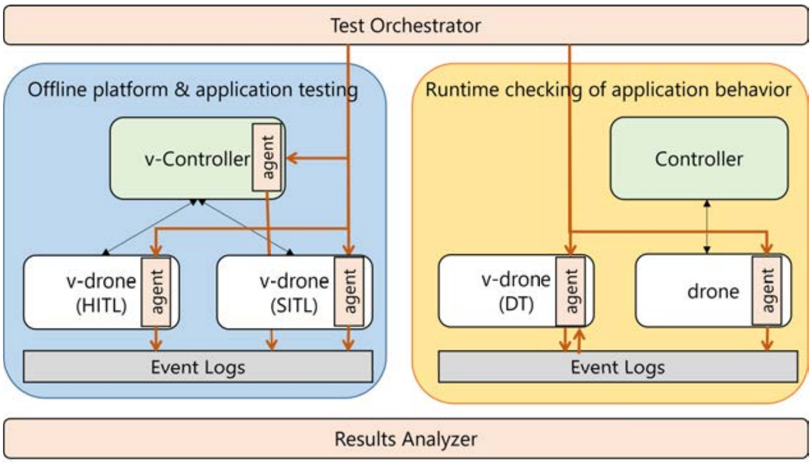


FIGURE 7.4: High-level view of the framework providing simulation and digital twin support for the PaaS platform

Real drones, v-drones and the v-Controller all feature an *agent* component. The agent exposes an interface that is used by the *Test Orchestrator* in order to properly configure the participating entities according to the needs of the specific test mode and objectives. Also, the agent is responsible for creating logs where different runtime events are recorded. These logs are sent to other agents, or they are stored in a central repository for further processing by the *Results Analyzer* in order to confirm expected behavior or detect deviations.

7.4.2 HITL and SITL configurations of v-drones

Platform and application testing can be performed at low cost and with zero risk using a simulated setup that consists of a v-Controller and one or more v-drones. In this case, the v-drone can be set to operate in a hardware-in-the-loop (HITL) or software-in-the-loop (SITL) configuration. Figure 7.5 illustrates the most typical simulation options (for brevity, the agent components are not shown).

In the HITL configuration, the Drone Runtime environment and the application both run on the hardware platform that is actually used in the real drone. The autopilot may also run on this hardware if it supports a HITL mode. Otherwise, it runs as a separate simulation entity (v-Autopilot) on the computing infrastructure of the testing facility, in which case it interacts with the rest of the drone software stack running on the target drone hardware through a proxy. Finally, it is possible to have a pure SITL configuration where the entire drone software stack (autopilot, Drone Runtime and application) runs on virtual hardware on the computing infrastructure.

The autopilot communicates with a simulator that implements the flight dynamics model for the drone. It provides artificial sensor data for the accelerometers, gyroscopes, magnetometers and barometers, which normally come from the drone's inertial measurement units (IMUs) during real operation. This sensor data is used by the autopilot to estimate the position, speed and acceleration of the drone, based on which control decisions are taken and corresponding commands are issued to the drone's actuators (e.g., motors). In the HITL configuration, the flight dynamics simulator runs as a separate entity, which is connected over serial or a fast network connection to the hardware where the drone software stack is running. In the SITL

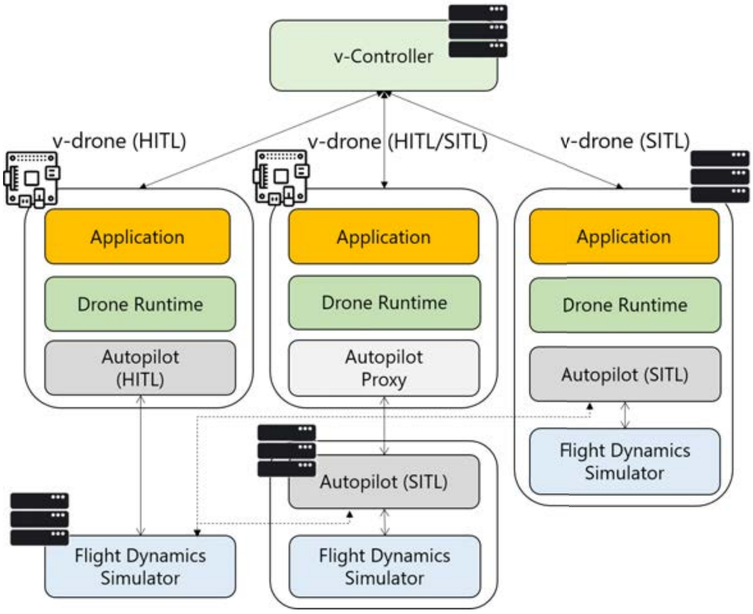


FIGURE 7.5: Simulated setup with HITL and SITL configurations

configuration, there is also the option of running the flight dynamics simulator together with the autopilot as a single simulation entity. The simulation framework is flexibly configurable allowing the user to choose the setup that is most appropriate for the specific testing objectives.

7.4.3 Offline platform testing

Using these simulation configurations, it is possible to test different aspects of the platform functionality. For this purpose, one typically designs suitable test applications. On the one hand, the resource allocation, application deployment and application execution functionality can be tested using “benign” applications that implement different types of missions (the PaaS supports specific templates). On the other hand, “malicious” applications, which intentionally attempt to perform actions that violate the flight plan and related restrictions, can be used to confirm proper operation of the platform’s monitoring and violation detection/handling mechanisms.

It is also possible to run different scenarios aimed at testing exceptional and critical situations. This can be achieved by instructing the v-Controller to issue commands to the drone, which change the flight plan, update restrictions, or even request an emergency landing, and then check that the platform behaves accordingly. Furthermore, one can experiment with intermittent connectivity and disconnected operation scenarios, by disrupting the (simulated) wireless communication between the v-drone and the v-Controller and observing whether the corresponding fail-safe actions are triggered.

7.4.4 Offline application testing

Once the platform has been tested to a satisfactory degree, the same simulation environment can be used to test and debug concrete applications in a practically open-ended fashion.

The usual objective of application testing is to verify that the application always behaves as expected. This can be achieved by designing and running a wide range of test scenarios, which combine different application configuration parameters, flight plans and restrictions. After each test run, the outcome that is recorded in the event logs can be compared to the expected application behavior.

Another typical objective of application testing is to confirm that the application does not consume more computing and communication resources than declared by the developer (as specified in the corresponding description). This can be achieved by properly configuring the resources that are allocated to the application on the v-drone as well as by observing the actual resource consumption during execution.

For such tests it typically suffices to use a pure SITL configuration of the v-drone. Of course, it is also possible to use a HITL configuration if it is desired to run the application (and the rest of the drone software stack) on the target hardware platform used in the real drone. As discussed, the flight dynamics simulator can be integrated with the autopilot or run separately, depending on the degree of fidelity required.

7.4.5 Digital twin for application checking at runtime

Despite elaborate offline testing, hidden hardware malfunctions or silent data corruption may still occur at runtime. It is thus highly desirable to be able to confirm that everything runs as expected during the real mission, ideally without involving a human observer.

This is achieved using a digital twin (DT) approach, a popular way of testing cyber-physical systems. More specifically, next to the drone that runs the application in the real world, the Drone Runtime and application also run on a v-drone. The v-drone is configured to operate in a special DT/SITL configuration, where the autopilot is replaced by a mockup and where the execution of the application and Drone Runtime occurs in conjunction with a *replay engine* that receives information from the real drone. Figure 7.6 shows a high-level illustration of the approach.

In the real drone, the agent component continuously records state-related information and streams it to the v-drone acting as its digital twin. This information includes the application startup event, the application requests that are received by the Drone Runtime (e.g., arm, takeoff, navigation or camera control commands), the corresponding (potentially adjusted) requests forwarded by the Drone Runtime to the autopilot, the Controller commands towards the Drone Runtime, as well as the telemetry logs that contain the autopilot's replies to received requests along with status and flight-related data published by the autopilot periodically. Each piece of information is timestamped using the drone's system clock. In addition, the requests of the application and the corresponding replies of the autopilot are tagged with increasing sequence numbers.

In the v-drone, the replay engine uses this information to drive the local execution. More specifically, the v-drone starts application execution with a pre-specified delay defined by the user (operator). The startup delay must be large enough to ensure a continuous flow of information from the real drone to the v-drone assuming a worst-case data transfer delay and jitter over the wireless connection, but also short enough to enable timely identification of execution inconsistencies. A startup delay in the order of a few seconds is typically sufficient for this purpose. Once application execution starts, the replay engine continuously checks the received information for

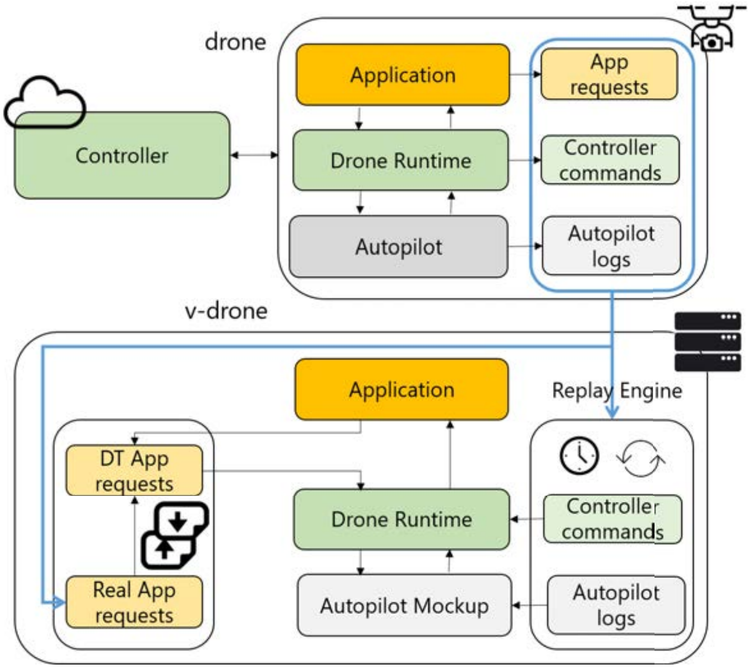


FIGURE 7.6: DT/SITL v-drone configuration

the chronologically next event, with a timestamp that is equal to the local clock value minus the startup delay. Autopilot status messages are sent through the mockup, whereas Controller commands are sent to the Drone Runtime.

The agent component of the digital twin intercepts application requests, retrieves their type and passed parameters, and compares them with the ones that were issued by the application running on the real drone. Obviously, requests with the same sequence number should be identical. If so, the agent forwards the application request to the Drone Runtime as usual, which, in turn, issues a request to the autopilot mockup that replies using the corresponding information supplied by the replay engine. Else, if the application request on the digital twin differs from the one that was generated on the real drone, which may be an indication of a possible malfunction, an alert is raised, and the digital twin terminates.

Note that this setup also makes the real application state easily accessible to testers, for instance, to inspect several performance indicators in depth when exploring optimizations. This can be done efficiently by accessing the internals of the digital twin in a direct way, instead of having to access the real drone over the wireless/mobile network, which can introduce a large delay but also consume valuable resources that should be reserved for more critical control operations.

7.5 Implementation

7.5.1 Platform as a service system

In our proof-of-concept implementation, in terms of software, we combine cloud-based services that provide the application deployment and application management functionality, with a drone-based environment that supports the controlled

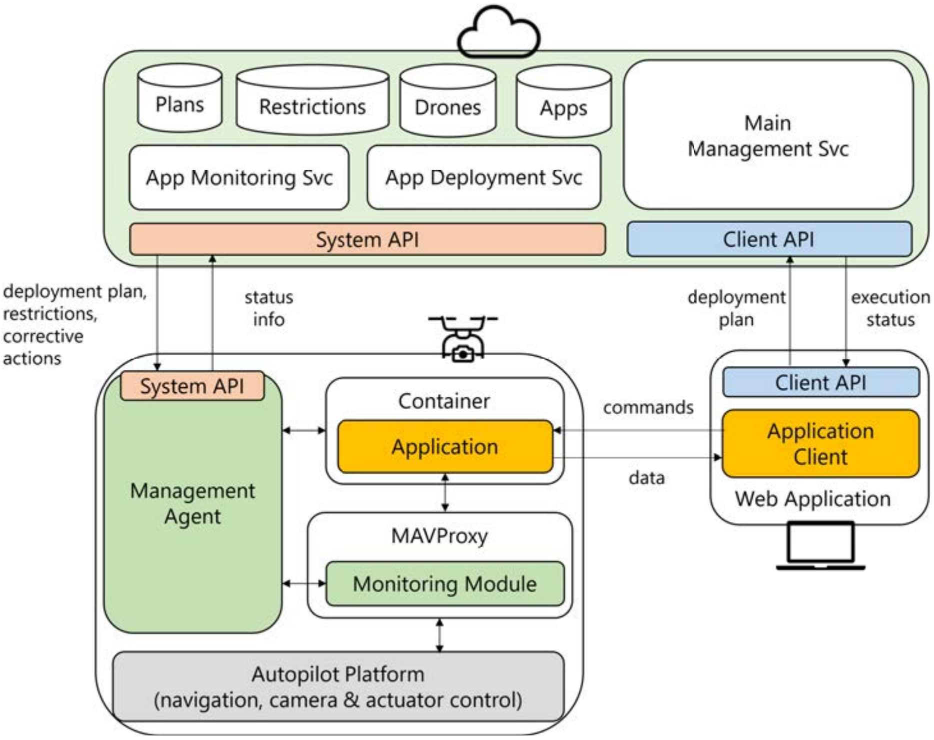


FIGURE 7.7: Software architecture of the system prototype and basic interactions

execution of the application and the enforcement of restrictions. The software architecture of our prototype, along with the mapping of the basic interactions between the main system components in this architecture, is presented in Figure 7.7.

Management controller

The management controller is the core management entity of our system. It runs on the cloud and is composed of a database/storage component and a set of distinct services. The storage component internally employs a NoSQL mongoDB database [117] to maintain (i) the application repository that includes the application executable images and descriptors, (ii) the drone repository with the descriptions for all the available drones, (iii) the restrictions repository that includes all the flight and sensor restrictions with their repair actions, and (iv) the deployment plans that are currently running or have been scheduled to run at a later point in time. These repositories are used by the different services of the management controller to perform the deployment checks. The services also implement the application deployment and drone monitoring processes.

The management controller exposes a RESTful API, which can be used by a client to create/submit an application deployment plan and retrieve the respective status updates. The provided primitives are listed in Table 7.1. The drones that can be used to deploy a given application are found through a matchmaking process, which compares the flight, computing, networking and sensor requirements from the respective application descriptor, with the corresponding resources and capabilities of the drones that are available.

TABLE 7.1: Client API of the management controller

Primitive	Description
/drones	Get available drones.
/drones/<int:id>	Get drone info.
/apps	Get available applications.
/apps/<int:id>	Get application info.
/apps/<int:id>/drones	Get drones for this application.
/app_plan	Create and submit plan.
/app_plan/<int:id>	Get, update, delete plan.
/app_status/<int:id>	Get application status.

The main management service is responsible for processing user requests and keeping track of the overall system status. When a deployment plan is submitted, it computes the flight area to be reserved based on the specified path/area and a safety radius/zone depending on the physical characteristics of the drone. The area of operation is then checked against the flight-restriction constraints that will be active during the specified period of application execution. The application's sensor requirements are also checked against the sensor restrictions for the area and time period of operation. The deployment plan is rejected if the sensors required by the application are restricted for the entire area and period of operation. Note that approval does not imply that the application can use the drone's sensors at any point in time during execution since the approved flight path/area may still overlap with areas where certain sensor restrictions apply.

If these checks are successful, the application deployment service is instructed to create a Dockerfile and build the respective Docker image, which is compatible with the computing resources of the selected drone and contains the application, the deployment plan and the related restrictions along with the respective corrective actions. We chose Docker containers as they offer portability and isolation while remaining lightweight is terms of image size and runtime resources. The image is then sent to the target drone via the internal system API.

Finally, the application monitoring service is responsible for monitoring the execution status of all deployed applications, and for reporting this information to the main management service. It also sends to the drones updates regarding the restrictions/repair actions if these change during the execution of the application, e.g., in the presence of an emergency event. The interaction between the drone and the application monitoring service occurs via the internal system API.

Drone environment

At the drone side, we have developed a prototype software environment that enables the monitoring and restriction of the application's behavior regarding the movement of the drone and the utilization of the onboard sensor equipment.

Given that the majority of drones utilize ARM embedded devices as companion boards, we opt for a Raspberry Pi 3 platform running the Linux-based Raspbian OS. For the autopilot software, we use the ArduPilot autopilot [7], which has been ported to many commercial and open-source hardware platforms and supports a variety of aerial vehicles ranging from fixed-wing to multicopters. Following what is common practice in the ArduPilot community, we use the PREEMPT_RT kernel patch [104], which supports real-time scheduling of the autopilot software with minimal latencies.

We consider applications that perform navigation operations, access sensors and issue actuation commands via the autopilot subsystem using MAVLink [112] messages. This covers a wide variety of applications built on top of the low-level C and python (pymavlink) MAVLink libraries, or higher-level APIs like MAVSDK [115], DroneKit [38] and ROS [149] using the MAVROS communication node. The communication between the application and the autopilot is done through MAVProxy [113], which supports multiplexing of the MAVLink protocol for multiple clients.

The management agent instantiates the application environment using the Docker image it receives from the management controller. During execution, it keeps track of and if needed enforces the respective deployment plan, which specifies the geofence/allowed area of operation, along with all the constraints for the relevant areas and the respective corrective actions. When the application terminates, the management agent performs all the necessary clean-ups, so that the next application can be deployed from a clean state.

To achieve the desired monitoring functionality, we have extended MAVProxy with an extra module that intercepts all the commands coming from the application and checks them before forwarding them to the ArduPilot subsystem. If a command violates one of the restrictions, the monitor takes the repair action defined for this type of violation in the deployment plan. If the application needs to be aborted, all its commands are blocked/ignored, and the application is forcefully terminated by stopping the container. Moreover, the monitor blocks a number of additional MAVLink commands, such as attempts to change the navigation mode, so as to ensure that the application runs only in the permitted, guided mode.

7.5.2 Simulation and Digital Twin

The design of the simulation framework is based on and extends AeroLoop [81], the simulation environment we presented in chapter 2, section 2.3. The v-Controller and the v-drone in the SITL and DT/SITL configurations are packaged as separate virtualized systems. Instead of virtual machines (VMs) used in the vanilla AeroLoop system, in this case, we use Linux Containers (LXDs) [110]. LXDs offer a good compromise between isolation and resource efficiency, as they provide operating system-level virtualization, which is more lightweight than full-fledged VMs, while offering a more complete virtual system (closer to a real drone) than Docker containers that share the same networking/storage stacks. For the HITL configuration of the v-drone, we support the Raspberry Pi 3 platform, which is the most popular companion board used in real drones.

The Test Orchestrator and Results Analyzer are implemented as Python libraries. The test scenarios are Python scripts that simply invoke these libraries. The agents are also implemented as Python programs, running as standalone processes with the respective interface being invoked via RPCs through the zerorpc library [128]. For logging, we utilize the standard Python logging facility, and event logs are processed by the Results Analyzer using standard Unix tools.

The Test Orchestrator creates and configures all simulation entities through the respective agents. It is also responsible for performing all the network configuration to enable the communication between the entities involved in a given test.

In simulation-based configurations, wireless networking is implemented using ns-3 [147]. For Wi-Fi channels, each simulated ns-3 node, called ghost node, utilizes the ns-3 TapBridge device, which is connected to each v-drone through a combination of network bridges and virtual network devices (see [81]). We also provide support for simulated LTE interfaces. This is achieved by introducing a high-bandwidth, low-latency CSMA link between a ghost node and the simulated LTE's UE node (see [132]). Finally, the v-drone agents continuously update the position of the respective ns-3 ghost nodes through a publish/subscribe scheme that is implemented using the ZeroMQ library [180]. In the DT/SITL configuration, the Test Orchestrator instructs the agents of the real and the v-drone to setup the physical connection that will be used to transfer the required information from the real drone to its digital twin.

Inside v-drones, the application is executed as a Docker container (this is how applications are packaged in the PaaS system). To this end, for v-drones running as LXDs we exploit the nested container functionality. Applications may perform different navigation operations, access sensors and issue actuation commands via the autopilot subsystem, using the MAVLink messaging protocol [112]. Various APIs offer MAVLink support, from low-level C and python (pymavlink) libraries to higher-level ones like DroneKit [38] and ROS [149] through the MAVROS communication node.

The autopilot used in v-drones is the latest stable version of ArduPilot [7], which is one of the most widely adopted autopilot stacks supporting a wide variety of aerial vehicles. In the HITL and SITL configurations, we use the pre-built binaries for the ARM and x86_64 architectures, respectively. The autopilot proxy in the hybrid HITL/SITL v-drone configuration is implemented using the MAVProxy [113] multiplexing tool, which forwards all messages of the Drone Runtime to the remote autopilot (running in SITL), and vice versa. Note that any autopilot that supports MAVLink and provides support for HITL or SITL simulation, such as PX4, could be easily integrated in our framework. The replay engine used in the DT/SITL configuration is implemented as a Python module, while the autopilot mockup utilizes the pymavlink library for receiving and sending MAVLink messages. The runtime inspection functionality is provided through standard system-level monitoring tools at two layers; at the digital twin (v-drone) to check resource utilization of the whole application container, e.g., using the docker stats command, and at the application layer to check the CPU/memory usage of specific processes, e.g., using top.

As mentioned in Section 7.4, the autopilot of a v-drone can be configured to use an integrated flight dynamics model or be connected with an external, higher fidelity simulator. ArduPilot has built-in support for the former option, while for the latter option we currently support Gazebo [77] through the corresponding plugin. Note that any other simulator that supports ArduPilot (e.g., AirSim [159]) could be used instead.

Finally, the simulation environment provides two options for simulated camera support to the application. When running the autopilot together with Gazebo, the ROS camera plugin can be utilized to publish the virtual camera stream of Gazebo to a ROS topic. Alternatively, we provide a custom virtual camera module for taking snapshots, which mimics the API of the Raspberry Pi camera module and returns images automatically extracted from a pre-configured database (see [81]).

7.6 Evaluation

The objective of our evaluation is to illustrate the main aspects of the functionality provided by both the platform as a service system for controlled application execution and the integrated testing support. To this end, we use indicative testing scenarios for the simulation environment and the digital twin configuration and focus on verifying the monitoring functionality and the enforcement of corrective actions in the presence of violations.

7.6.1 Offline simulation experiments

Setup. Listing 5 shows the setup sequence for an offline test using a v-drone and a v-Controller that communicate via Wi-Fi. In a nutshell the steps are to: (i) create a v-Controller entity (line 1); (ii) instantiate a v-drone with identifier "vDrone-1", which is set in SITL mode using the ArduPilot autopilot and Gazebo as the flight dynamics simulator, with system resources (in memory and disk) as indicated in a configuration file (lines 2-5); (iii) set the drone's camera source from gazebo that returns satellite imagery from a static map plugin (line 6); (iv) set a waypoint application as a pre-installed application at the v-drone (as opposed to deploying it dynamically via the v-Controller) and specify its parameters (takeoff altitude and waypoints) (lines 7-8); (v) set the approved flight plan, which also specifies restrictions and the respective corrective actions (line 9); (vi) set the start location / home position of the v-drone (line 10); (vii) create a Wi-Fi network (lines 11-12); (viii) activate logging at all available levels (lines 13-14); and (ix) start the application (line 15).

Listing 5 Setup and start offline test

```
1  controller = vController()
2  drone = vDrone("vDrone-1", SITL,
3                 autopilot=Ardupilot,
4                 fdm=gazebo,
5                 setup=config.min_resources)
6  drone.set_camera(gazebo-satellite)
7  drone.set_app("waypoint_app",
8                params=config.app_params)
9  drone.set_plan(config.flight_plan)
10 drone.set_pos(config.home_pos)
11 network = NetSim(config.net_wifi)
12 network.add_participants(controller, drone)
13 drone.set_logs(app, runtime, autopilot)
14 controller.set_logs(runtime)
15 drone.start_app()
```

Scenario. The test application, implemented with DroneKit, issues a sequence of waypoint-based navigation commands. The next command is issued once the previous target location has been reached or a timeout has expired. In parallel, the application can perform sensing by instructing the camera sensor to capture still images. Figure 7.8 depicts the test scenario in the city of Volos. The green waypoints (WP1, WP2, WP3, WP4), placed at the four roundabouts near the port of the city, indicate the locations that are specified by the user as part of the deployment plan. These define the path that the application is expected to follow, i.e., WP1→WP2→WP3→WP4. The light green shaded area indicates the (geofenced) area of operation where the drone is allowed to fly. This is generated by the system automatically based on the physical characteristics of the drone that runs the

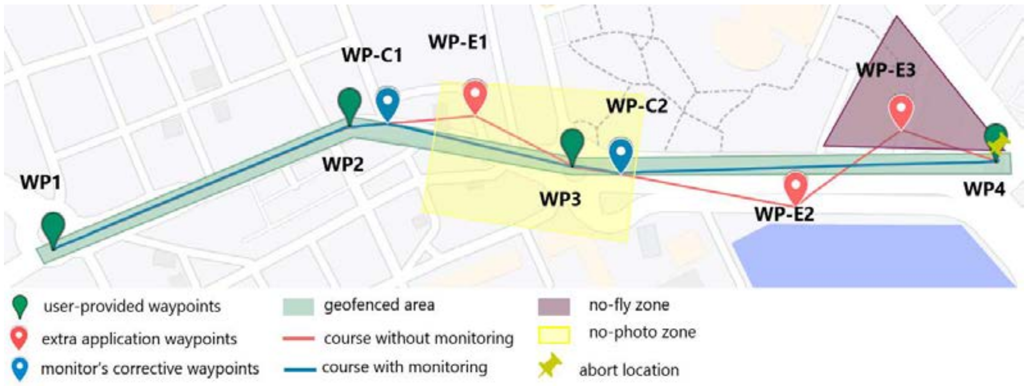


FIGURE 7.8: Simulation experiment with the monitoring mechanism disabled (red line) vs enabled (blue line)

application. In this case, a safety perimeter of a few meters is applied along the straight-line segments connecting the waypoints of the deployment plan. The dark purple shaded triangle (over the city hall) indicates a zone where flying is strictly prohibited. The light-yellow region indicates a zone where the usage of a camera is not permitted. The yellow pin, placed at the same location as WP4, denotes the target location in case the application is aborted.

Flight restrictions. To verify the functionality of our system, the test application is programmed to deviate from the course declared in the deployment plan. The waypoints of the *extra* navigation commands that are issued at runtime are marked in Figure 7.8 with the red waypoints (WP-E1, WP-E2, WP-E3). Thus, the actual course that is followed by the application when our monitoring mechanism is disabled is WP1→WP2→WP-E1→WP3→WP-E2→WP-E3→WP4, depicted by the red line.

The blue line in Figure 7.8 shows the result for the same test scenario with our monitoring mechanism enabled. Namely, when the application issues navigation commands to the locations WP-E1 and WP-E2, which are outside of the expected area of operation, these commands are corrected so that the drone remains within the allowed flight path. More specifically, the monitor instructs the drone to move towards the intersection of the path it would have followed based on the application's request and the borders of the geofenced area (WP-C1 and WP-C2, respectively). Later on, when the application targets location WP-E3, which is not only outside the expected area of operation but also inside a no-fly zone, it performs a major violation. Consequently, the monitor decides to stop the application and instructs the drone to move to the specified abort location (WP4) for landing. Thus, the resulting course is WP1→WP2→WP-C1→WP3→WP-C2→WP4.

Camera sensor restrictions. To verify the enforcement of sensor-related restrictions, we run the test application with the correct waypoints, i.e., those specified in the deployment plan. In addition, the application is programmed to take photos periodically, every 4 seconds, while the drone is moving between WP2 and WP4. As it can be seen in Figure 7.8, this course trespasses the no-photo zone. Again, we run two test scenarios, with our monitoring mechanism disabled and enabled, respectively. Figure 7.9 shows the sequence of images produced in the former scenario. The images that are crossed-out (entire second row and the first image in the third row) are taken by the application while flying above the no-photo zone. These images are automatically suppressed by the monitoring module when our mechanism

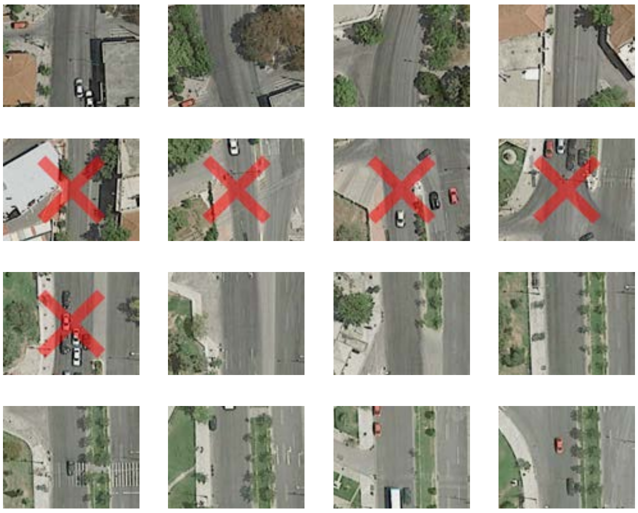


FIGURE 7.9: Sequence of images captured by the application between WP2 and WP4 (crossed out images are suppressed with monitoring mechanism enabled)

is enabled and the application receives an error when trying to access the camera while the drone is above these locations.

Results analysis. Apart from this kind of functional evaluation, there are also other checks that can be performed during simulation execution or after its completion. Indicative examples are given in Listing 6. These checks can have the form of queries or assertions. In the first case, specific data are returned, such as information regarding application violations, the size of data sent, or the average system resources used by the application during execution (lines 1-3), in order to, e.g., compare them by hand with the expected outcome in case of successful completion (line 4). In the latter case, checks directly examine the application status or the existence of a violation of a specific type (lines 5-6).

Listing 6 Analyze other test results

```
1 info = get_violations("vDrone-1")
2 data = get_data_sent("vDrone-1", level=app)
3 perf = get_app_resources("vDrone-1")
4 compare(perf, config.expected_perf)
5 check_app_status("vDrone-1", SUCCESS)
6 check_violations("vDrone-1", type='flight_path')
```

7.6.2 Runtime real-world testing

Setup. For the runtime checking using the digital twin configuration in the real-world experiment, we use the custom-made hexacopter presented in chapter 2, section 2.2, shown in Figure 7.10. Besides the autopilot board with the basic flight-related sensors and actuators, the drone features a Raspberry Pi 3 onboard computer with the Drone Runtime environment of the PaaS platform, which is used for the application deployment and execution. The onboard computer is also used to run the agent of the testing framework.

Listing 7 shows the script for setting up such a test. In a first step, an object representing the real drone is created and its configuration, i.e., application, parameters and approved flight plan, is retrieved (lines 1-3). Then, a virtual drone is configured

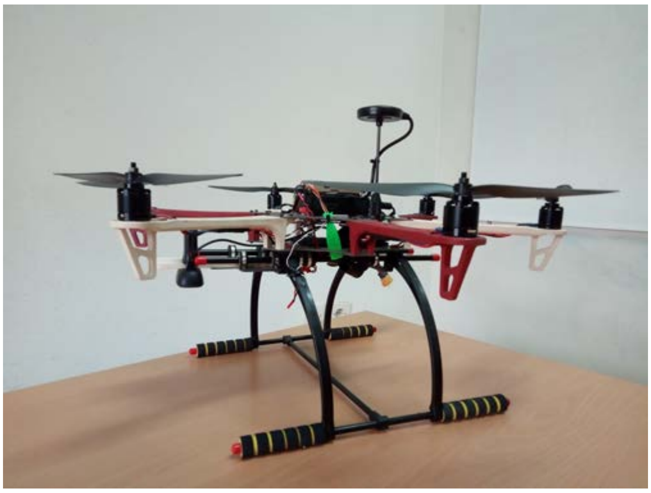


FIGURE 7.10: Hexacopter used in the real-world experiment for the runtime checking

in digital twin mode and is initialized using the application/flight plan configuration of the real drone. Finally, the two entities are connected with each other in order to enable the streaming of all the state-related data from the real drone to the digital twin (line 7).

Listing 7 Setup and start online test

```
1 real_drone = realDrone("drone-1")
2 test_config = real_drone.get_config(app, params, plan)
3 vdrone = vDrone("vDrone-1", DT)
4 vdrone.set_app(test_config.app,
5                params=test_config.params)
6 vdrone.set_plan(test_config.plan)
7 connect_drone_with_dt("drone-1", "vDrone-1")
```

Scenario. The test application is a Python program that arms the drone, takes off to WP1 (at a height of 10 meters), goes to waypoint WP2, next moves to waypoint WP3, returns to initial location WP1 and lands. This is done by issuing corresponding commands to the autopilot via DroneKit. Figure 7.11 depicts the test scenarios performed in the open field.

Flight restrictions. To demonstrate the detection of unexpected deviations at runtime, we modify the application that runs on the real drone so that it executes a slightly different mission. More specifically, after reaching WP2, it issues a request to navigate towards waypoint WP-R3, as shown in the dark green path depicted in Figure 7.11. In contrast, the application running (correctly) on the digital twin requests a movement to WP3 (as it should), which corresponds to the white line. This deviation is detected and raises an alarm. The digital twin terminates at this point, and it is up to the tester/operator to decide which action to take concerning the real drone.

Results analysis. The API for comparing the behavior of the application running on the real drone vs on its digital twin is straightforward, as shown in Listing 8.

During execution one may continuously compare the requests issued by the application on the digital twin with those issued by the application on the real drone, or simply list these requests in an explicit way (lines 1-3). Also, as long as the execution

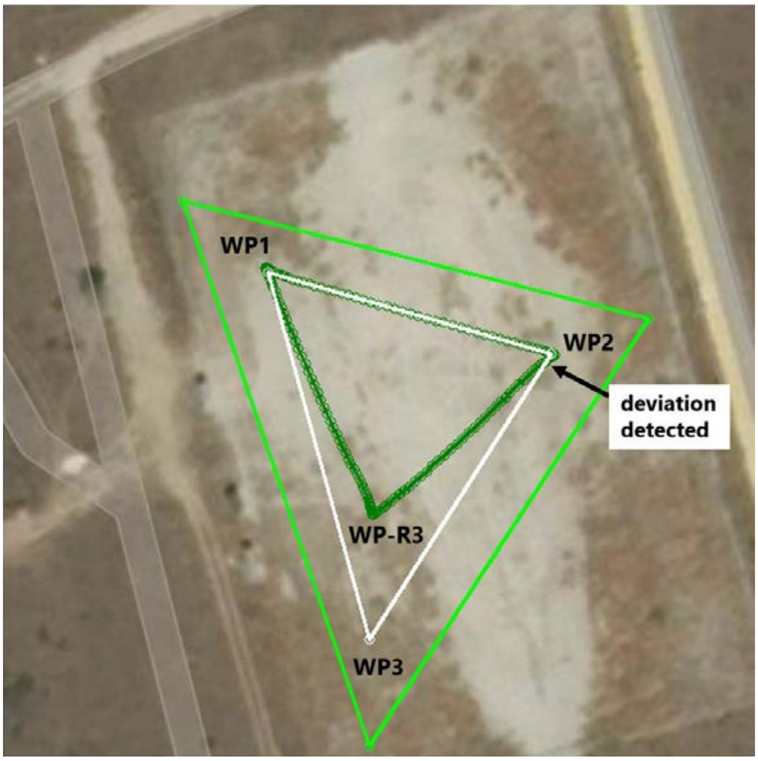


FIGURE 7.11: Runtime checking experiment

Listing 8 Check application behavior at runtime

```
1 vdrone.compare_app_requests()
2 vdrone.list_app_requests(source=real)
3 vdrone.list_app_requests(source=dt)
4 vdrone.dt_exec("docker stats --no-stream")
5 vdrone.app_exec("top -b -d10 -n2 >top_results.txt")
```

of the digital twin is identical to that of the real drone, one can inspect several application performance aspects simply by querying the digital twin, e.g., by executing commands to check the application's container state, or even the detailed resource usage of specific processes (lines 4-5).

Chapter 8

Orchestration of Distributed Drone-Edge-Cloud Applications

8.1 Contributions and Outline

In this chapter, we present *Fractus*[1], an orchestration framework for the automated deployment and connectivity management of distributed, component-based applications across the entire system continuum, including drones, edge nodes and the cloud. We view such a framework as an in important step towards next-generation drone applications that have to incorporate data from various edge and cloud sources and need to interact with the edge in an efficient, mission-aware fashion.

Our main contributions are the following:

- We present the design of a complete orchestration framework, which automatically deploys and interconnects application components across the drone-edge-cloud continuum.
- We introduce intuitive abstractions for describing the application's placement and communication requirements.
- Resource allocation and component deployment is requirements-aware, based on the geographical area where the application will use the drone.
- Connectivity between application components is managed transparently, exploiting ad hoc local networking opportunities between drone and edge nodes that host interacting components.
- Critical drone and edge functions, such as mobility and sensor operations, are accessed by the application in a controlled way, through interfaces that offer secure, safety- and privacy-preserving operations, driven by structured policies that can be specified and updated by the relevant stakeholders in a flexible way.
- We discuss a concrete implementation of Fractus, based on Kubernetes [94], the most widely used container orchestration platform, which we extend in several ways to achieve the desired functionality.

[1] Fractus are small cloud fragments with irregular patterns, found under an ambient cloud base. They change constantly, often forming and dissipating rapidly.

- Through an extensive evaluation, we illustrate the provided functionality via real field tests and a simulation environment, and we show that our implementation decreases the development effort and incurs an acceptable resource footprint even for the constrained platforms found on drones and edge nodes.

The rest of the chapter is organized as follows. Section 8.2 motivates our work and gives an application example through which we illustrate the concept of Fractus. Section 8.3 introduces the concept and overall design of Fractus. Section 8.4 presents the descriptors Fractus uses internally for the representation of the various resources, as well as the provided abstractions for the specification of the user requirements. Section 8.5 focuses on the application deployment process. Section 8.6 describes the design of the network management functionality. Section 8.7 provides more details regarding the policy-based service access to sensor and mobility resources. Section 8.8 discusses the most important aspects of our implementation. Finally, Section 8.9 presents the experiments performed to evaluate several aspects of our prototype regarding the resource/performance overhead and the provided functionality.

8.2 Motivation and Concept

Next-generation drone applications will not be just about navigating the drone and collecting data via its onboard sensors. They will incorporate data from different sources and will include heavyweight data processing that cannot be performed solely by the drone. For instance, surveillance and monitoring applications may combine aerial and ground-level image feeds and involve different image and big data processing tasks. Also, last-mile delivery applications may need to run different signal and image processing jobs in order to land and/or place cargo in a receptor with high accuracy. Even though it is possible to run some of the processing tasks in the cloud, in some cases it can be more beneficial to offload them to the edge, e.g., to minimize latency or reduce the amount of data transfers over metered 4/5G connections.

However, to achieve this flexibility one has to: (i) locate edge nodes in the drone's operational area capable of hosting the desired application components; (ii) schedule application components on such nodes according to the application's needs; (iii) establish direct connectivity between drone and edge nodes whenever it is deemed beneficial. Currently, all these issues are tackled manually, targeting specific setups, which makes application deployment inflexible in adapting to different environments or changing conditions. Our vision is to provide a framework where the user only provides the application's components and requirements, and all the above issues are taken care of in a transparent way. This allows developers to focus on the core application logic, while the application components can run unmodified in different setups.

Take as a concrete example a road traffic monitoring application that uses both an airborne camera on a drone and ground cameras on static edge nodes. The upper part of Figure 8.1 shows an indicative component structure: the *Navigator* guides the drone so that it follows the desired path along specific waypoints to provide the required aerial coverage; the *AerialViewer* captures and filters the images of the drone camera, while the *GroundViewer* performs a similar operation for a static camera on an edge node; the *ImageChecker* is responsible for the core image processing part of the application; finally, the results are forwarded to the *DataStore* that saves them

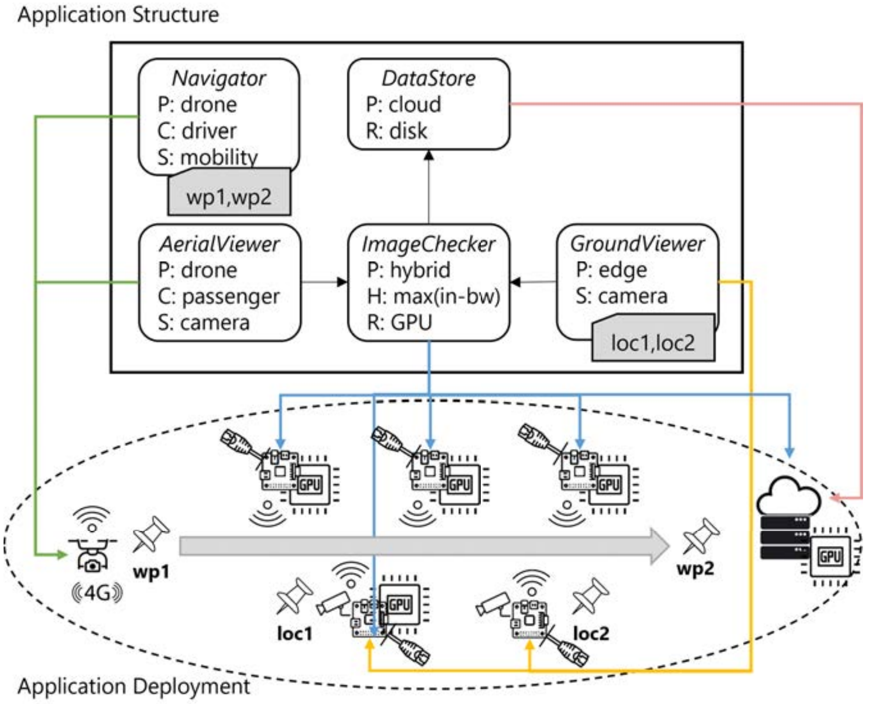


FIGURE 8.1: Application structure and indicative deployment based on the requirements of each component

in persistent storage, from where they can be made available to other applications. The lower part of Figure 8.1 shows the system infrastructure. Note that the drone and edge nodes have ad hoc wireless networking interfaces. In addition, the drone is assumed to have stable 4/5G Internet connectivity, whereas all edge nodes have a wired connection to the Internet.

What Fractus proposes is that the deployment is driven by placement (P), class (C), heuristic (H), service (S) and resource (R) specifications that accompany the code of each component (see Section 8.4.3). The arrows directed from the application structure in Figure 8.1 to the nodes of the system infrastructure illustrate such an indicative deployment. More specifically, the *Navigator* and *AerialViewer* run on a drone where they access the mobility and camera service, respectively. The *GroundViewer* is instantiated multiple times on camera-equipped edge nodes at specified locations of interest. The *ImageChecker* is instantiated multiple times on nodes that have GPU computing resources. Apart from creating a base instance in the cloud, additional instances of the *ImageChecker* may be hosted on edge nodes along the path of the drone, which can interact with the *AerialViewer* component in a direct way via WiFi instead of using a 4/5G Internet connection. In a similar vein, additional *ImageChecker* instances may be placed on edge nodes hosting or being close to *GroundViewer* components. During application execution, the selection of the actively interacting instance of *ImageChecker* component instance that actually communicates with the *AerialViewer* and the *GroundViewer* components is based on the heuristic targeting the maximization of the bandwidth for ingress traffic, as specified by the application. Finally, the *DataStore* component is placed in a cloud node with ample storage.

To make such a model of operation attractive, we follow the cloud's paradigm and embrace multitenancy to improve resource usage and lower costs of operation. However, even though the efficient sharing of computing resources like CPU and memory can be achieved using well-known virtualization and resource-limiting practices, the multitenant usage of drones along with their sensors in the edge is more complicated as applications from different users may have different access rights to sensor data and to the drone's mobility. To this end, Fractus regulates access to such critical resources through the combination of the specified application requirements and external safety and privacy restrictions provided by the relevant stakeholders. Also, the resulting fine-grained access rights are enforced in a hard way by the system itself, without depending on the goodwill or skills of the application developer.

To continue the above example, assume a different application for monitoring the air pollution due to car traffic, which includes a *GasMeasure* component for measuring gas emissions at both the air and the road level and a *GasStore* cloud component for storing the measured concentration of these gases. In this case, Fractus may deploy the *GasMeasure* component on the drone and the edge nodes that are already being used by the road traffic monitoring application, providing access to the CO/CO2 on-board sensors. In the same spirit, the component of yet another application that analyses the movement of pedestrians can be deployed on the same drone/edge nodes too and be given access to the on-board cameras being used by the *AerialViewer/GroundViewer* components of the road traffic application. However, for these components, the system might activate (possibly different) privacy-preserving policies for the images retrieved or even block image retrieval in certain areas. In addition, Fractus can restrict the *Navigator* component of the road traffic application so that it cannot direct the drone into a no-fly zone or descent below a certain altitude.

8.3 Fractus Overview

Fractus is designed to support the orchestrated deployment and interconnection of next-generation drone applications that seamlessly integrate drone-edge-cloud resources while ensuring safety and privacy.

In particular, the main design goals are to build a system that can enable (i) the automated resource discovery-allocation and application scheduling based on specified deployment needs, (ii) the efficient management of networking resources, leveraging ad hoc networking technologies to support direct device-to-device communication between components running on drones and edge nodes, and (iii) the controlled access of sensor, actuator and mobility resources on drones and edge nodes through proper service interfaces and policies.

To achieve the desired flexibility, Fractus focuses on applications that adopt a microservice approach, consisting of a collection of relatively small, independently deployable and loosely coupled software entities (microservices) that communicate with each other through well-defined interfaces. The individual application components are packaged as separate containers, providing isolation and resource limiting in an efficient and lightweight manner, which is important for the resource constrained computing platforms of drones and low-cost edge nodes.

Figure 8.2 shows the main entities of Fractus, along with an indicative deployment of the road traffic and pollution monitoring applications discussed in Section 8.2. The user submits, through an API server, a structured description for the application, and Fractus takes care of the necessary component deployment and networking. As

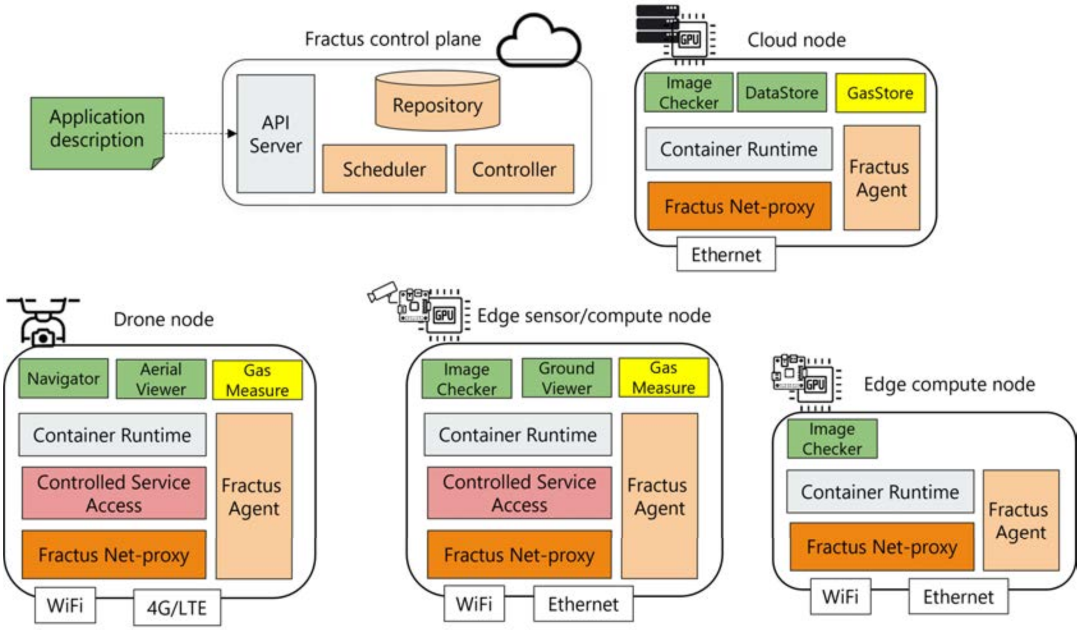


FIGURE 8.2: Fractus architecture and indicative deployment

can be seen in the figure, the components of a single application are distributed in the drone-edge-cloud continuum, and a single drone, edge or cloud node can host components from different applications.

In the following sections, we discuss the key aspects of Fractus. We focus mainly on the drone- and edge-related deployment and networking aspects. Also, we do not discuss the algorithmic parts of the implementation in great detail; we note, however, that the respective code is in the form of plug-ins that can be replaced with more optimized versions, if desired.

8.4 Resource Descriptions

To achieve the desired functionality in an automated way, several pieces of information regarding the available system resources must be explicitly captured and maintained up to date in an efficient manner. Also, the application description submitted by the user must specify in a clear and intuitive way the respective requirements for each application component.

8.4.1 Node descriptions

For each drone and edge node, the Fractus Repository keeps record of both its static properties and the dynamic operation parameters. Listing 9 and Listing 10 provide snippets of the descriptions for a drone and an edge node, respectively. Static properties include physical features (e.g., size and weight of drones, the location of edge nodes), the sensing and mobility resources offered to the application through corresponding services (see Section 8.7), service-specific capabilities (e.g., maximum speed and flight time, for mobility) and the available ad hoc networking interfaces with the pairing information needed to establish communication links with other nodes.

Dynamic parameters include the available computing resources (e.g., CPU, memory, storage). In addition, for drones, the descriptions include status information (inactive, charging, taking-off, flying, landing), battery characteristics (used to estimate the remaining flight time) and current position. Further, the Repository includes information about the location of depot stations where drones are kept when not in use or when they need to charge their batteries.

Listing 9 Drone description

```yaml
type: drone
name:  drone-1
mobilityType: quadcopter
class: C0
# drone static parameters
physicalFeatures:
dimensions: {height: 100, length: 60, width: 50}
mtom: 1230    # maximum takeoff mass, in grams
networkResources:
- interface: 4G
- interface: WiFi
pairingInfo: {mode: adhoc,  networkId: fractus-adhoc-1}
systemServices:
mobility:
  autopilot: ardupilot
  maxAltitude: 80   # in metres
  maxFlightTime: 15 # in minutes
  maxSpeed: 15      # in m/s
  methods: [Arm, Takeoff, Goto, GetPosition, Hover,
            SetSpeed, GetSpeed, ReturnToLaunch, Land]
# drone dynamic parameters
status: flying
location: [lat, lon, alt]
battery: {current: 28.9, remaining_level: 82, voltage: 10.25}
```

Listing 10 Edge node description

```yaml
type: edge
name:  edge-1
location: [lat, lon, alt]
networkResources:
- interface: Ethernet
- interface: WiFi
pairingInfo: {mode: adhoc,  networkId: fractus-adhoc-2}
systemServices:
camera:
  model: RPi Camera v2
  resolution: 3280x2464
  sensorType: RGB
  methods: [CaptureImage, RetrieveImage, ListImages]
```

During a node's registration with the Fractus system, the respective Agent submits the node description to the Repository. At runtime, the Agent updates the values of dynamic parameters in response to requests of the Fractus Controller or periodically during application execution.

8.4.2 Policy descriptions

In a similar manner, the Fractus Repository stores so-called access policies. These can be submitted by relevant authorities (e.g., civil aviation, municipalities) and drone/edge node providers, to regulate the degree to which applications can access a given sensing/mobility resource through the corresponding service, as well as to filter the replies returned from the service. A policy can optionally have a selector expression specifying the areas, nodes and applications for which the policy holds.

The currently supported access policies are summarized in Table 8.1, while Listing 11 presents some indicative examples. An *AccessControl* policy (line 1) acts as an authorization mechanism granting or forbidding access to specific service calls. *LimitControl* policies (lines 6 and 12) are used to set upper and/or lower limits to different quantities related to the mobility service of a drone, such as speed and flying altitude, while *GeofenceControl* (line 20) determines how to handle application requests to move in an unauthorized area or move out of an authorized area. The supported actions vary from completely ignoring the request, to adjusting the request within the allowed bounds, to more radical measures like forcing an immediate landing. Finally, a *PrivacyControl* policy can be used to control how privacy is preserved with respect to a specific sensor. More concretely, in the case of a camera, this concerns the captured images, which can be altered in order to return a black image to the application, blur the whole image, detect and blur faces in the image, or remove the background.

Listing 11 Policy descriptions

```
1  - type: AccessControl # method access
2    kind: denial # forbid access
3    methods: [CaptureImage, RetrieveImage]
4    selector: # for all drones/edge nodes in a given area
5      area: region1
6  - type: LimitControl # speed limit
7    kind: speed
8    upperLimit: 10.0
9    action: enforce # adopt bound
10   selector: # for all drones flying above a certain area
11     area: region2
12 - type: LimitControl # altitude limits
13   kind: altitude
14   upperLimit: 50.0
15   lowerLimit: 5.0
16   action: enforce # adopt corresponding bound
17   selector: # for a given area and certain drone classes
18     area: region2
19     drone: { key: class, operator: In, values: [C2, C3]}
20 - type: GeofenceControl # mobility limits
21   kind: exclusion # prevent from entering
22   action: land # take over and land the drone
23   selector: # for drones exceeding a specific weight
24     area: region3
25     drone: { key: mtom, operator: Gt, value: 2000}
26 - type: PrivacyControl   # for sensors
27   kind: camera # for the camera
28   action: blurFaces # use suitable filter
29   selector: # for all drones/edge nodes in a given area
30     area: region3
```

8.4.3 Application and component descriptions

Application descriptions specify for each component the desired placement, the required resources/services related with static features and dynamic parameters, and

TABLE 8.1: Generic and resource-specific service policies

Policy	Description
GENERIC	
`AccessControl`	grant/forbid access to service call(s)
MOBILITY	
`LimitControl`	restrict speed and altitude values
`GeofenceControl`	correct a spatial violation
CAMERA	
`PrivacyControl`	filter image retrieval

the ingress/egress interactions with other components. Next, we focus on the parts that drive the most important functionality of Fractus for the drone and edge nodes. As a tangible example, Listing 12 provides snippets of the component descriptions accompanying the application discussed in Section 8.2.

The basic component placement options are "drone" as for the *Navigator* and *AerialViewer*, "static edge" as for the *GroundViewer*, "hybrid edge-cloud" as for the *ImageChecker* meaning that multiple instances can be deployed on different edge nodes and the cloud, or "cloud" as for the *DataStore* (not shown in the listing). More specialized information needs to be provided depending on the placement option. For instance, cloud components can specify the desired number of replicas, hybrid components have to specify the heuristic for selecting the currently active instance (discussed below), while edge components may have specific location requirements. Drone and edge components need to specify the mobility and sensing service primitives they will invoke at runtime.

Drone components declare the required degree of mobility/navigation control on the drone, by classifying themselves as a "driver" or "passenger". A driver component may request full control for the entire flight procedure or partial control at specific locations. Respectively, it may specify a single control point as the start location (which can also be left open) or several ones where it needs to take over. Each control point specifies the type of navigation that will be used, which can be a path defined via a sequence of waypoints, or a region defined as a bounding polygon. For instance, the *Navigator* component of the traffic monitoring application is a driver that requires full control and applies path-based navigation. In contrast, passenger components (*AerialViewer*) take advantage of a driver and can use the drone's sensor services without having to perform any explicit navigation/mobility operations.

If the application does not have a driver, one of the passenger components has to specify the points of interest to be visited by the drone, whether these points need to be visited in a given order (thereby constituting a path) and whether access to the drone's services is required only at these points or throughput the specified path, as shown in Listing 13. All other passenger components (if any) implicitly inherit this specification. Fractus will then try to find a suitable driver that can accommodate these passengers (see Section 8.5). Notably, Fractus assumes that drones have obstacle avoidance capability as part of the basic autopilot stack and does not deal with such low-level issues.

A static edge component (*GroundViewer*) explicitly declares the areas of deployment (center location and radius) and the required number of instances for each of them

Listing 12 Simplified description of the traffic monitoring application

```
- component: Navigator # drone driver
  placement: drone
  class: driver
  services:
  - service: mobility
    methods: [Arm, Takeoff, Goto, SetSpeed, Land]
  control: full
  controlPoints:
  - point: homeLocation
    navigation: pathBased
    path: [wp1, wp2, ...]
- component: AerialViewer # drone passenger
  placement: drone
  class: passenger
  services:
  - service: camera
    methods: [CaptureImage, RetrieveImage]
  egress: [ImageChecker]
- component: GroundViewer # static edge
  placement: edge
  locations: [{loc1, radius1, 1}, {loc2, radius2, "*"} ]
  services:
  - service: camera
    methods: [CaptureImage, RetrieveImage]
  egress: [ImageChecker]
- component: ImageChecker # hybrid cloud-edge
  placement: hybrid
  heuristic:
    kind: max
    metric: ingress-bandwidth
  resources: [GPU]
  ingress: [AerialViewer, GroundViewer]
  egress: [DataStore]
  hybridTo: [AerialViewer, GroundViewer]
- component: DataStore # cloud component
  placement: cloud
  replicas: 3
  ingress: [ImageChecker]
```

(* stands for "as many as possible"). Once an instance of a static component is deployed on a node, it remains there in an active state during the entire lifetime of the application.

Hybrid cloud-edge components (*ImageChecker*) are instantiated at least once in the cloud. Furthermore, additional instances can be created as close as possible to a subset of the interacting (ingress and egress) drone/edge components, specified by the "hybridTo" directive, to maximize direct communication opportunities (e.g., over WiFi). Thus, the placement of the extra instances is determined by the operation area of the drone and the location of the edge nodes hosting the static components, respectively. A different instance of a hybrid component may be created for each instance of a static edge interacting component (*GroundViewer*). Moreover, for drone-based interacting components (*AerialViewer*), multiple instances of a hybrid component can be deployed on several edge nodes to achieve good coverage that maximizes direct communication opportunities. During application execution, Fractus transparently directs application traffic to or from only one so-called *target* hybrid

Listing 13 Description of passenger component with navigation information

```
component: AerialViewer
placement: drone
class: passenger
services:
- service: camera
  methods: [CaptureImage, RetrieveImage]
egress: [ImageChecker]
pointsOfInterest:
  points: [wp2.1, wp2.2, wp2.3]
  ordered: yes # visit in the order listed (path)
  serviceAccess: path # for the entire path
```

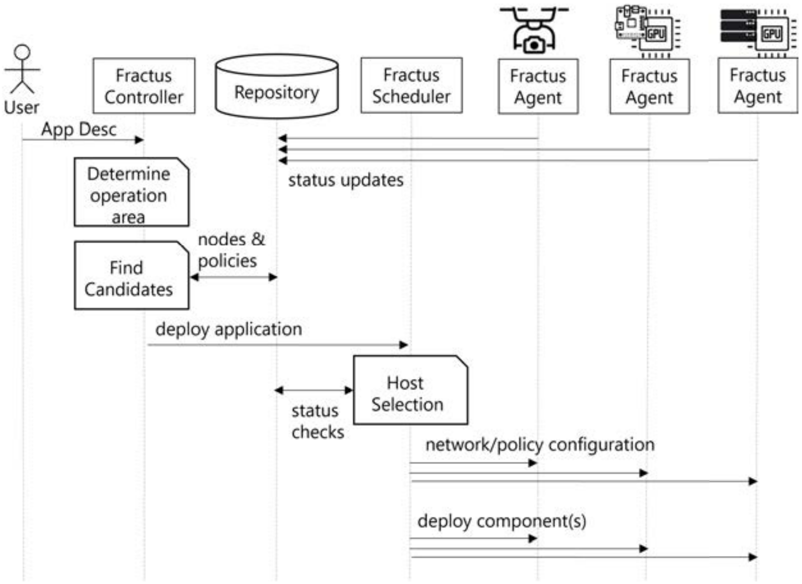


FIGURE 8.3: Sequence of application deployment

instance at any point in time (see Section 8.6). The selection is based on corresponding declarations in the application description. In the current prototype, the application may opt for maximum bandwidth or minimum latency for ingress and/or egress flows, or always prefer direct links vs communication over 4/5G Internet connections. Fractus also provides suitable hooks so that new heuristics can be easily integrated as plugins into the platform. To eliminate the need for state migration we currently consider stateless hybrid components, which is a common practice in microservice architectures.

8.5 Application Deployment

When an application description is submitted, Fractus performs several steps to deploy its components in the drone-edge-cloud continuum, illustrated in Figure 8.3. Next, we discuss the deployment process in more detail, focusing on the key aspects that concern deployment on the drone and edge nodes. More specifically, we discuss how Fractus (i) establishes the area where the drone will be used by the application (Algorithm 5), (ii) uses this information to find candidates for hosting the application's drone, static edge and hybrid cloud-edge components (Algorithms 6, 7, 8), (iii) selects the specific hosts where each component instance will be deployed (Algorithms 9,10).

Algorithm 5 Calculation of drone operation area

```
Input: $app$                                          ▷ application components descriptions
Output: $area, poi$                         ▷ drone area(s) of operation, passenger PoI

area, poi ← ∅,∅
if $app.driver \neq NULL$ then                       ▷ area based on driver navigation info
    for each $ctrlp \in app.driver.ctrlPoints$ do
        $area \leftarrow area \cup$ calcArea($ctrlp.navigationInfo$)
    end for
else                                           ▷ area based on passenger points of interest
    $psgr \leftarrow (c \in app.passengerComponents \wedge c.poi \neq \emptyset)$
    $area \leftarrow$ calcArea($pasgr.poi$)
    $poi \leftarrow psgr.poi$
end if
```

Area of drone operation. If the application has a driver, the area of operation is calculated using the navigation information (path or a wider region) for each declared control point. Else, if the application only has passenger components, the area of operation results from the respective points of interest.

Algorithm 6 Find candidates for hosting the drone components

```
Input: $app, poi$
Output: $cand_{drone}$

if $app.driver \neq NULL$ then                                      ▷ find unused drones
    $startPoint \leftarrow$ first($app.driver.ctrlPoints$).$start$
    if $app.driver.control =$ full then                          ▷ drones at specific depot
        $cand \leftarrow$ dronesInDepotAt($startPoint$)
    else                                                        ▷ drones at nearby depots
        $cand \leftarrow$ dronesInDepotCloseTo($startPoint$)
    end if
else                              ▷ find suitable used drones and nearby unused drones
    $cand^{used} \leftarrow$ dronesUsedVistingSame($poi$)
    $cand^{idle} \leftarrow$ dronesInDepotCloseTo(first($poi$))
    $cand \leftarrow cand^{used} \cup cand^{idle}$
end if
$cand_{drone} \leftarrow$ sortByStatusAndFlightTime($cand$)
```

Host candidates for drone components. If the application includes a driver component with full navigation control, the drones considered as candidates are those found in depot stations at the specified start location. If the driver does not specify the start point or requires partial control, candidate drones are the ones in depots within a (configurable) radius from the first control point of the application driver. In this case, Fractus provides a "chauffeur" (system driver component) that will fly the drone from the depot to the first control point of the application driver as well as between any other control points. The chauffeur by default follows the shortest path, which is a straight line in the absence of obstacles.

If the application only has passengers, Fractus considers as candidates drones already allocated to other applications, provided the points of interest lie within the driver's specified path and are visited in the required sequence (if one is specified). In addition, unused drones close to the path's start are also considered as candidates; in this case, similar to above, Fractus provides a chauffeur that will navigate the drone through all points of interest, following the shortest path, as mentioned above.

Finally, the drone candidates are ranked according to their flight readiness level, energy level and estimated (remaining) flight time, in the spirit of [133] [21]. For applications without an own driver, an extra scoring rule is applied to give preference to drones that are already used for other applications, instead of employing new ones as this will lead to additional take-offs/landings and further loads the airspace.

Algorithm 7 Find candidates for hosting the static edge components

```
Input: $app$
Output: $cand_{edge}$

for each $s \in app.edgeComponents$ do
    for each $l \in s.locations$ do
        $cand \leftarrow \text{nodesWithin}(l.area)$
        $cand_{stat}[s][l] \leftarrow \text{sortByDist}(cand, l.center)$
    end for
end for
```

Host candidates for static edge components. Finding host candidates for each static edge component of the application, is more straightforward. For each such component and desired deployment location, all nodes within the corresponding area specified in the application description are considered as candidates. The candidates are then sorted in increasing order of their distance to the center location of the specified deployment area.

Algorithm 8 Find candidates for hosting the hybrid components at the edge

```
Input: $app, area, cand_{edge}$
Output: $cand^{edge}_{hybrid}, cand^{drone}_{hybrid}$

for each $p \in app.hybridComponents$ do
    for each $s \in p.hybridToEdgeComponents$ do
        for each $l \in s.locations$ do
            for each $n \in cand_{edge}[s][l]$ do
                $cand \leftarrow \text{nodesCloseTo}(n.pos)$
                $cand^{edge}_{hybrid}[p][s][l][n] \leftarrow \text{sortByDist}(cand, n.pos)$
            end for
        end for
    end for
    if $p.hybridToDroneComponents \neq \emptyset$ then
        $cand \leftarrow \text{nodesCloseTo}(area)$
        $cand^{drone}_{hybrid}[p] \leftarrow \text{sortByCoverage}(cand, area)$
    end if
end for
```

Host candidates for hybrid components. Apart from the cloud, there are additional candidates for hosting the edge instances of a hybrid component. For each interacting static edge component, the candidates for hosting the hybrid component are the nodes close to each one of the candidates (including the candidates themselves) previously found for the static edge component and each of the specified deployment locations. These candidates are then sorted based on their distance to the candidate hosts of the static edge component and their ability for direct communication via a suitable wireless interface. Also, if the hybrid component interacts with some drone component(s), the candidates for hosting the former are all edge nodes close to the calculated area of drone operation. In this case, the candidates are ranked in descending order of their coverage of the operation area through direct wireless communication.

Algorithm 9 Host selection for drone and static edge components

```
Input: $app, cand_{drone}, cand_{edge}$
Output $h_{drone}, h_{edge}$

// select host for all drone components
check($cand_{drone}$)                                         ▷ availability & resources
$h_{drone} \leftarrow cand_{drone}[0]$

// select hosts(s) for each static edge component and location
for each $s \in app.edgeComponents$ do
    for each $l \in s.locations$ do
        check($cand_{edge}[p][l]$)                            ▷ availability & resources
        $nofhosts \leftarrow$ max($l.instances$,len($cand_{edge}[p][l]$))
        $h_{edge}[s][l] \leftarrow cand_{edge}[p][l][i : nofhosts]$
    end for
end for
```

Host selection. Based on the candidates found through the previous location-based filtering process, the Controller produces a deployment plan, which is passed to the Fractus Scheduler. In turn, the Scheduler selects the hosts for placing the different instances of the application components. This is done by checking the status and resource availability of every candidate in the Repository (discarding those not meeting the respective requirements), and then picking the best of the remaining candidates.

Algorithm 10 Host selection for hybrid components at the edge

```
Input: $app, area, cand^{edge}_{hybrid}, cand^{drone}_{hybrid}, h_{edge}$
Output $h^{edge}_{hybrid}, h^{drone}_{hybrid}$

for each $p \in app.hybridComponents$ do
    for each $s \in p.hybridToEdgeComponents$ do
        for each $l \in s.locations$ do
            for each $n \in h_{edge}[s][l]$ do
                check($cand_{edge}[p][l][n]$)                 ▷ availability & resources
                $h^{edge}_{hybrid}[p][s][l][n] \leftarrow cand^{edge}_{hybrid}[p][s][l][n][0]$
            end for
        end for
    end for
    if $p.hybridToDroneComponents \neq \emptyset$ then
        $h^{drone}_{hybrid}[p] \leftarrow \emptyset$
        $covered \leftarrow \emptyset$
        check($cand^{drone}_{hybrid}[p]$)                     ▷ availability & resources
        while $\frac{covered}{area} < MAX\_COV \wedge cand^{drone}_{hybrid}[p] \neq \emptyset$ do
            $i \leftarrow$ getMaxCov($cand^{drone}_{hybrid}[p], covered, area$)
            $n \leftarrow$ rmv($cand^{drone}_{hybrid}[p][i]$)
            $append(h^{drone}_{hybrid}[p], n)$
        end while
    end if
end for
```

The best drone candidate is picked to host all the application's drone components. For each static edge component, the number of selected hosts depends on the number of desired instances per specified deployment area and the number of suitable candidates found for it, giving preference to the best candidates. For each hybrid component that interacts with a static edge component, at most one host is picked for each instance of the static edge component (the best among the candidates found). For each hybrid component interacting with drone components, multiple hosts are selected in order to achieve good coverage of the drone's operation area through direct wireless communication. To this end, a greedy approach is employed, by iteratively picking the next candidate that maximizes the coverage until a (configurable) percentage of the operation area is covered or there are no more candidates.

Component deployment. As a last step, the Scheduler deploys the instances of each application component on the selected hosts. Note that the application can be functional even if only the base cloud instance is created for a hybrid component even though this may not be ideal from a scalability and/or communication perspective. For instance, the traffic monitoring application could work with a single cloud *ImageChecker* instance, which could be accessed by the *AerialViewer* on the drone and all *GroundViewer* instances on camera-equipped edge nodes. But, of course, it would be better to have multiple instances of the *ImageChecker* on/near each such edge node as well as on nodes near the drone's path to be able to achieve the specified deployment requirements in the best possible way (see Figure 8.1).

8.6 Network and Application Flow Management

Drones, edge nodes and cloud nodes have a stable Internet-based communication channel that is used for the system-level interactions with the cloud control plane. For drones, this channel is maintained over cellular 4/5G, whereas edge and cloud nodes employ a wired Ethernet connection. Application-level communication between hybrid components placed in the cloud and their interacting drone and static edge components always takes place over this channel. In addition, at runtime, Fractus exploits ad hoc wireless interfaces to form in the background local and possibly ephemeral network links between edge nodes hosting instances of hybrid components and the hosts (drones/edge nodes) of their interacting components. This is done to examine alternative communication paths for interacting drone-hybrid and edge-hybrid components and select the target instance of the hybrid component that is optimal according to application requirements. The networking management is carried out, transparently to the application, by the Fractus Controller and the network proxy components that run on each node, as shown in Figure 8.4. Next, we discuss the main design aspects.

Addressing. The routing of application-level traffic is done through virtual IP addresses (VIPs). Fractus assigns a VIP to each service-providing application component that is accessed by other client components according to the ingress/egress relationships in the application description. In case the same service component is instantiated multiple times (e.g., *ImageChecker*), a single VIP is assigned to all instances.

Direct communication links. At node startup, the Connectivity Manager on the drone and edge nodes creates a separate Interface Controller for each networking interface except the one used for the default communication channel. These Interface Controllers are responsible for performing the technology-specific network creation, discovery and connection operations needed to establish a direct IP connection at the physical layer (besides WiFi, it is possible to include other technologies, e.g., BLE).

During the component deployment phase, the Scheduler sends to the Agents of the respective nodes the network configuration information generated by the Fractus

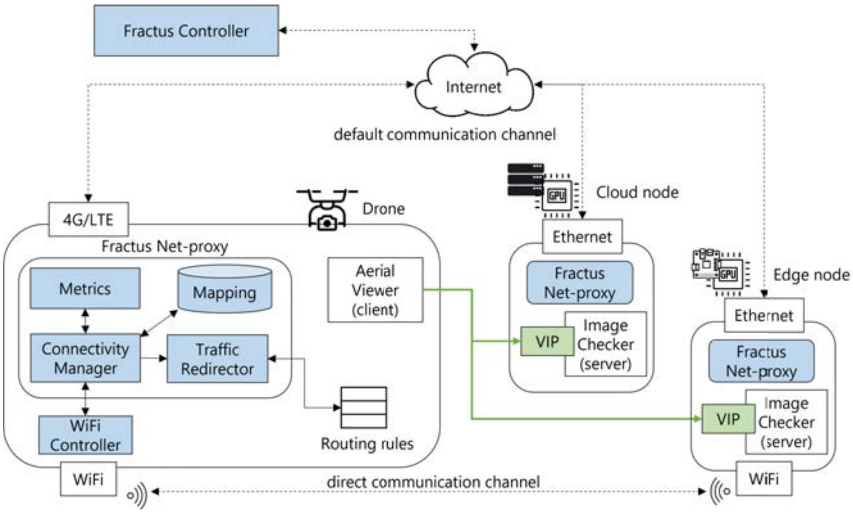


FIGURE 8.4: Requirements-aware networking of interacting components in the traffic monitoring application

Controller based on information stored in the Repository for each drone and edge node. This configuration is stored in the local Mapping directory and essentially consists of entries in the form <*name, VIP, netInfo*>, where *name* is the unique identifier of a component instance, *VIP* is the virtual IP address assigned to a server component, and *netInfo* is the required binding information for the ad hoc networking technology to be used.

At application startup, when there are hybrid components interacting with a static edge component (e.g., *ImageChecker* with *GroundViewer*), the Fractus Controller instructs the creation of the direct communication link between the edge node hosting the static component and the closest edge node hosting an instance of the hybrid component. Upon the reception of such a command, the Connectivity Manager on the static edge component host retrieves from the Mapping directory the relative *netInfo* and requests the respective Interface Controller to connect with the interacting node. This connection persists throughout the application execution.

During application execution, when there are multiple edge nodes hosting instances of a hybrid component that either provides a service (e.g., *ImageChecker* to *AerialViewer*), or invokes a service provided by a drone component, the Fractus Controller instructs the net-proxy component of the drone to create a direct communication link with a specific edge node. In this case, the decision of the node is based on the geographical proximity between the drone and the edge nodes, the specified range of the ad hoc networking technology and other navigation-related parameters. In path-based navigation, current drone position and speed combined with the next waypoint are used to proactively select the next node. In region-based navigation, the drone's position, speed and direction can be used to estimate its trajectory and select a new node accordingly. Upon the reception of such a command, the Connectivity Manager requests the respective Interface Controller to leave the current local network (if any) and initiate connection establishment to the newly selected one.

Target instance of hybrid component. The target instance for hybrid components interacting with drone or static edge components is selected by the Fractus Controller and can change dynamically during application execution. At application startup, the target is set to the base instance located in the cloud, whereas the instances at edge nodes are in an inactive state, meaning that they neither create nor receive any application traffic.

When a direct link is created, the Fractus Controller gets informed and depending on the corresponding application requirements, it may decide to switch immediately the target instance or start specific performance measurements. More specifically, if the heuristic is to minimize communication costs, it means that direct links are always favored over 4G/Internet communication; thus, the corresponding traffic flows at the involved nodes are redirected without any further delay. Otherwise, the Controller sends a request to the involved nodes to start running and reporting the related metrics. This process is performed by the Metrics component, which may need to measure the bandwidth of ingress and egress traffic, network latency and link quality for the available communication paths. The Fractus Controller monitors the received measurements and switches the target instance whenever deemed beneficial. Note that for drones such a switch may also need to take place due to moving away from the range of the edge node hosting the target instance, in which case the base instance located in the cloud is selected until a better option arises.

Once a switching decision is made, the Connectivity Managers on the drone/edge node and the node hosting the new target instance exchange routing information regarding the interacting components to make the redirection in a coordinated way. More specifically, the Traffic Redirector on the host of the server component changes the local routing rules to enable ingress traffic through the selected communication channel, whereas its counterpart at the host of the client component enables the corresponding egress traffic through this channel. Then, the previous target instance is set in an inactive state by blocking any incoming/outgoing interactions.

8.7 Access of Sensor and Mobility Services

Fractus exposes to the application the sensing and mobility capabilities of drone and edge nodes in the form of node-local system services, which are accessed through well-defined APIs. Our prototype supports two basic services: (i) the mobility service for controlling the movement and navigation of the drone and (ii) the camera service for recording and accessing images.

Table 8.2 lists the respective APIs. Note that the camera service provides separate functions for capturing and retrieving images since both of them are relatively time-consuming operations in typical computing platforms used in drones and edge nodes and may not be needed to perform them concurrently (e.g., in case the images are not processed immediately). Besides the methods for the mobility and camera services, Fractus also provides the `GetControl` method for the application to block until the next control point is reached (instead of constantly polling the drone's location), and the `ReturnControl` method for voluntarily releasing control once the application has completed its task within a given control area.

Internally, Fractus handles service method invocations based on a so-called *policy configuration* that is produced for each application and component. This configuration consists of refined policies that are generated by the Controller based on the component's declared service usage, the calculated application operation area and location requirements, the concrete candidate hosts and the relevant system policies stored in the Repository (Section 8.4.2).

TABLE 8.2: Basic system services API

Method	Description
GENERIC	
`GetControl`	block until next control point
`ReturnControl`	explicitly return control
MOBILITY	
`Arm/Disarm`	arm/disarm drone's motors
`Takeoff/Land`	take off to an altitude/land
`SetSpeed/GetSpeed`	set/get speed
`Goto`	move to specific location
`Hover`	maintain position
`GetPosition`	get current position
CAMERA	
`CaptureImage`	capture a still image
`RetrieveImage`	retrieve an image

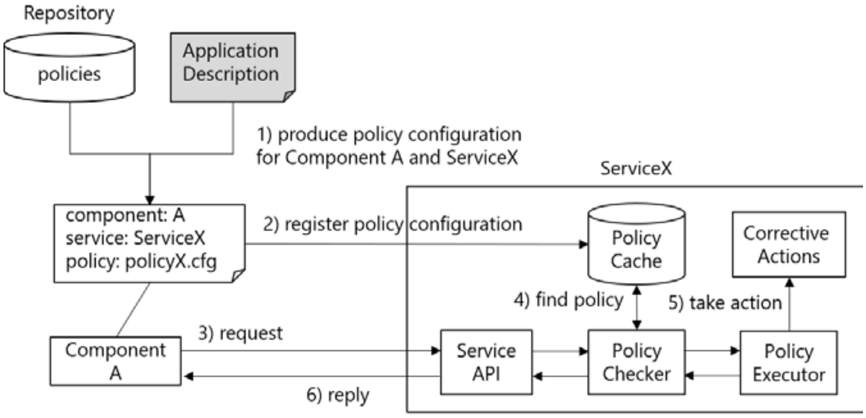


FIGURE 8.5: Policy-based service access

Taking the application example discussed in Section 8.2, the generated AccessControl policies for the *Navigator* component in Listing 12 concern the methods of the mobility service, while for the *AerialViewer* and the *GroundViewer* they concern the methods of the camera service. Also, for the *Navigator* and *AerialViewer* method access is restricted to the operation area of the drone. Furthermore, GeofenceControl policies are generated for the *Navigator* to restrict the location-related parameters of the `Goto` mobility method within the operation area and to force a landing if the application issues a call that attempts to enter a no-fly zone. In addition, the parameters of the `SetSpeed` and `Goto` methods of the mobility service can be further restricted as to the flight speed and altitude, based on respective LimitControl policies for the operation area of the application or the specific type of drone used to run the application. Finally, PrivacyControl policies that apply to certain areas may further restrict the results returned by the RetrieveImage method of the camera service to the *AerialViewer* and *GroundViewer* components.

The policy configurations generated by the Controller for the application's components are included in the deployment plan that is handed over to the Fractus Scheduler. In turn, the Scheduler sends this information to the host Agents together with the images of the application components to be instantiated there. The system services on drones and edge nodes have a specific internal structure, which is used to handle the service invocations performed by the application at runtime, as illustrated in Figure 8.5. In a nutshell, the policies for each application component are extracted from its configuration (before the component starts running) and are registered in a cache. Service requests are intercepted by the Policy Checker which queries the cache for policies matching the current location context, the service method and the component that performs the invocation. Depending on the outcome, the Policy Checker may handle the request as usual, reject it, or apply a corrective action. In the latter case, the request in forwarded to the Policy Executor, which provides implementations for specific corrective actions, and the produced reply is returned to the application component. Besides the service call-specific content, replies also carry status information describing the corrective action taken (if any) so that the application can be aware of this and proceed accordingly if needed. Finally, when a component stops its execution, all related policies are removed from the cache.

8.8 Implementation

The Fractus prototype is based on the Kubernetes container orchestration platform. We use k3s [71], a lightweight and fully conformant production-ready Kubernetes distribution for edge environments, which we extend in various ways to provide the required functionality.

The descriptions of drones, edge nodes, applications and policies are introduced as custom resources [90] through Custom Resource Definitions (CRDs). The actual instances are stored in the persistence store of the cluster as custom objects whose lifecycle management (CRUD operations) takes place through the Kubernetes API Server [88].

The Fractus Controller is implemented as a custom controller following the operator pattern [93]: it registers to the API Server as event listener for new Fractus application objects, on such events transforms the application description to corresponding Kubernetes objects, submits these objects for deployment and monitors their execution. The location-based scoping of candidate nodes is performed by annotating them with application- and component-specific labels. During the deployment plan creation, each component instance is transformed to a pod, the smallest deployable unit in Kubernetes, and its main deployment requirements are specified through *nodeSelector* and *nodeAffinity* rules that use these labels. The Fractus Scheduler extends the filtering and scoring phases of the upstream Kubernetes scheduler and is responsible for the binding of the application components (pods) to the selected hosts. We use the default set of predicates for filtering, and the location-based exclusion of candidate nodes is achieved via the built-in label-matching rules.

The Fractus Agent is a custom daemon, running in parallel with the node's Kubelet. At startup, it communicates with the API Server to register a new node object and add Fractus-specific labels corresponding to its features and starts the system services (mobility/camera). Then, during application execution, it informs Kubernetes about resource availability and the current values of its dynamic properties by updating the respective node object description in the persistence store through the Kubernetes API. The camera and mobility services are implemented as gRPC services using SSL/TLS authentication. Thus, for each component that accesses them the Fractus Scheduler has to add the respective credentials to its pod.

For the default communication channel, we employ the Flannel Container Networking Interface plugin [48] which configures a layer 3 IPv4 overlay network between all cluster nodes. The VIP address of service-providing components is mapped to Kubernetes Service resources [96]. For each such Service, the Kubernetes network proxy (kube-proxy) running at each node, uses the Linux kernel *netfilter* framework and installs iptables rules to capture traffic to the service's VIP/port and redirect it as needed through the overlay network. During the selection of the target instance of a hybrid component, the Fractus Controller uses *Network Policies* [92] that control this traffic flow to isolate the other instances through suitable ingress/egress rules.

The Fractus Network proxy essentially expands the capabilities of the Kubernetes network proxy and in our prototype focuses on exploiting WiFi. The WiFi Controller supports discovery using both ad hoc and infrastructure modes. After the successful pairing at the physical layer, if the Fractus Controller selects an edge node as target, the Traffic Redirector (on both hosts) creates a mapping between the VIP, the address in the Kubernetes overlay network of each local server application component and a new IP address that is created for it in the direct IP network. Then, corresponding iptables rules are inserted in the *prerouting* and *postrouting* chains of the NAT table, capturing packets sent to the direct network IP addresses and changing their destination to the cluster-wide component addresses. Subsequently, this information is exchanged between the two hosts so that the other side also inserts corresponding iptables rules that capture egress traffic to each such virtual IP and redirect it to the direct network.

8.9 Evaluation

Our evaluation seeks to answer the following three main questions:

- Is Fractus practical in terms of the overhead introduced to the resource-constrained drone/edge nodes?
- What are the tangible benefits regarding the policy-based access to resources in a real-world setup?
- What are the gains of the requirements-aware deployment and network management?

To address these questions, we evaluate the various resource overheads of Fractus using a hardware-based testbed comprising of typical drone and embedded edge nodes hardware platforms. Then, we use these platforms to showcase part of the provided drone-related functionality through field experimentation. Finally, we demonstrate more of the provided functionality and related benefits using a simulated setup that also allows us to study the deployment overhead of our approach at scale.

Both field and lab experiments, make use of the hardware testbed setup, shown in Figure 8.6, which is based on the system presented in chapter 2, section 2.2. In particular, the drone used is a custom-made hexacopter with a CUAV V5 nano autopilot board [168] running the ArduPilot autopilot software [7] (Copter 4.0). The drone has as a companion board a Raspberry Pi 3 Model B [145] with a quad-core ARM Cortex A53 processor (@1.2 GHz, 1GB RAM) running the Raspberry Pi OS Buster. The RPi is connected to the autopilot board over serial and runs the Fractus software. The mobility service employs DroneKit [38], which communicates with

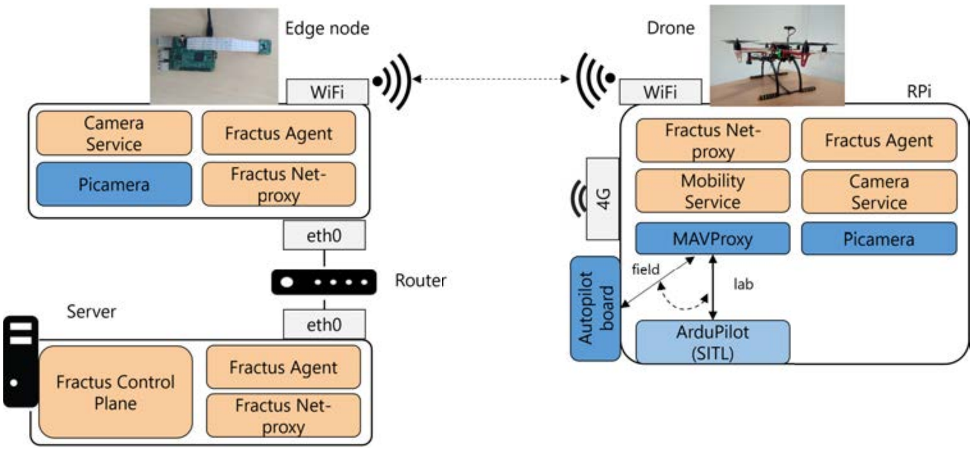


FIGURE 8.6: Hardware testbed

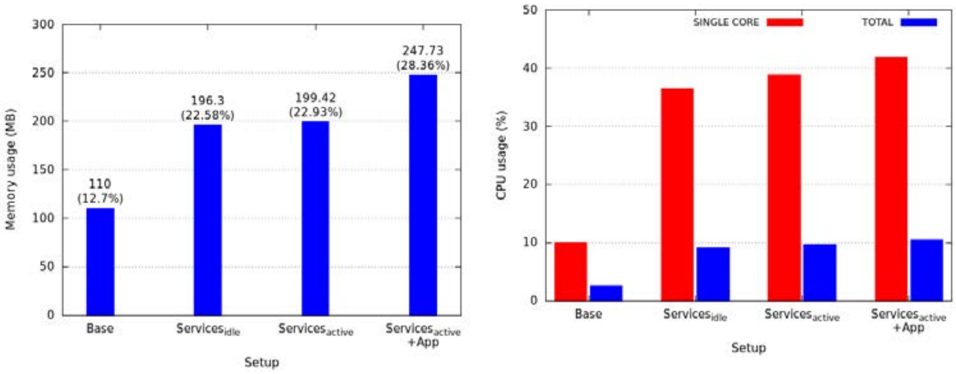
the autopilot through MAVProxy [113]. The drone is also equipped with an 8MP Raspberry Pi Camera Module 2 [144], which is accessed by the camera service using the picamera library [129]. The drone communicates with the Fractus control plane and edge/cloud nodes over Internet via a 4G/LTE USB modem, whereas RPi's WiFi interface is used for direct communication.

In the following sections, at first, we describe the specific system configurations and then we present indicative evaluation experiments and results.

8.9.1 Resource usage and performance overheads

Setup. Our lab measurements focus on capturing key performance overheads of Fractus on the RPi without having to fly the drone. Thus, the Kubernetes cluster consists of the drone, which uses the software-in-the-loop (SITL) ArduPilot configuration [8] (see Figure 8.6), an edge node and a cloud node, all these interconnected inside a VPN over Internet. The cloud node is placed on the server running the Fractus control plane and as an edge node, we employ another RPi that uses its Ethernet interface for the default communication and can also interact with the drone directly over WiFi.

Disk. The Kubernetes-related functions provided by k3s amount to just about 50MB since old and non-essential code is removed and the respective agents (kubelet, kube-proxy, flannel agent) are packaged as a single binary. Similarly, the Fractus-related agents run natively in the host node, while for the mobility and camera service we opt for containerized images in order to manage them through Kubernetes. These are based on the officially provided slim variant of the python images for ARM and include all the required packages, libraries and their dependencies, taking together a bit more than 1GB, which is less than 4% of the RPi's 29GB storage space of the microSD card we use in our setup. In particular, the mobility service image is 450MB including gRPC with protocol buffers for the service definition and data serialization, DroneKit for accessing the autopilot, and Shapely/GEOS [52] for spatial/geometry-related operations, while the camera service in addition includes the OpenCV library for the image filtering functionality along its dependencies resulting in 680MB. Thus, it leaves ample storage space for the applications components, for which the only software requirement, is to include gRPC in their container



(A) Memory usage

(B) CPU usage as a percentage of a single core and of all the available CPUs

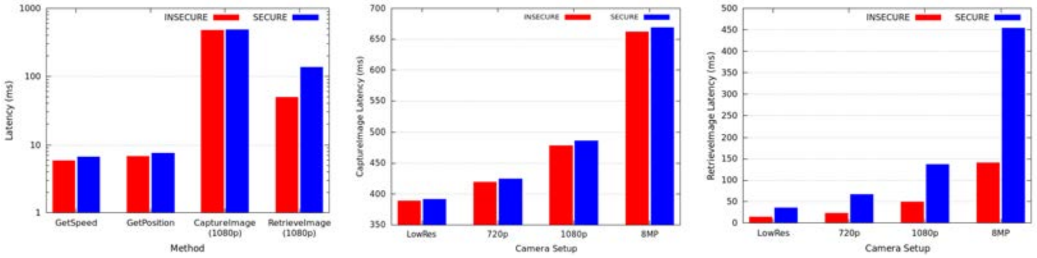
FIGURE 8.7: Memory and CPU usage in different setups

image in case they need to access the mobility and camera services. An indicative image size for such python-based components invoking any of these services is around 300MB.

Memory & CPU. We measure the memory and CPU usage for different setups, gradually considering more functionality. At first, we consider only the system-level agents: the k3s and Fractus agents (*Base*). Then, we add the mobility and camera services when they are idle waiting for requests ($Services_{idle}$). Next, we use an application with two drone components: a driver that invokes in a loop the Goto primitive of the mobility service and then periodically invokes the GetPosition primitive until the target location is reached, and a passenger that periodically invokes the CaptureImage and RetrieveImage primitives of the camera service and stores the retrieved images to disk. No service access policies are applied in this case. We measure the total memory/CPU requirements, at the system level ($Services_{active}$) and including the application ($Services_{active} + App$).

Figure 8.7a shows the memory usage in MB in the different setups. We also report the percentage with respect to the total available memory. Note that while the RPi ships with 1GB of RAM, this is split between the CPU running the host OS and a separate GPU used for camera-related processing. In order for the camera service to have acceptable performance, we give 128MB to the GPU, leaving 872MB available for the rest of the system. Thus, the percentages reported are with respect to the host OS available memory. For the *Base* setup, the usage of 110MB ($\sim$ 13% of the available RAM) is mainly related to the container lifecycle management operations. The mobility and camera services use 86MB when idle; usage increases only slightly when serving application requests. Finally, the two application components combined use another 50MB of RAM, resulting in a total of 250MB ($\sim$ 29% of the available RAM). This leaves more than 600 MB free for other application components.

Figure 8.7 depicts the CPU usage as a percentage of a single core and in relation with the CPU capacity of the four cores that are available in the RPi, for the different setups. The *Base* setup takes on average 10% (2.5% of the total available processing power). The addition of the mobility and camera services introduces another 26.5% of a core when idle, mainly due to background threads of MAVProxy that periodically poll the autopilot. The CPU usage of Fractus in a single core is increased merely by 2% during application execution when these services become active and



(A) Various service methods (log scaled)

(B) CaptureImage method for different camera setups

(C) RetrieveImage method for different camera setups

FIGURE 8.8: Invocation latency for SECURE vs INSECURE setups

start handling the invocations of the application components. Finally, the two application components combined utilize less than 4% of a single core (1% of the total CPU capacity). The total CPU usage is slightly more than 10% of the total CPU capacity, about 40% of a single CPU, leaving plenty of room for the application components to include more complex logic as well as data processing.

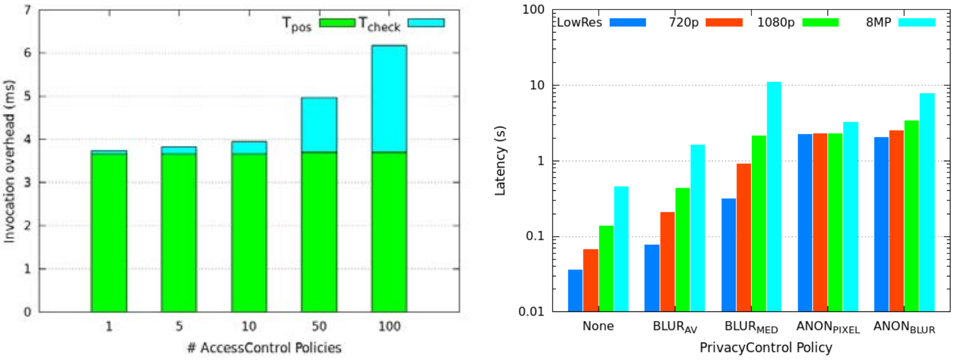
Service authentication mechanism. To determine the overhead of the authentication/security mechanism used in Fractus for the communication between the services and the client components, we measure the service call invocation latency for various service primitives when the SSL/TLS mechanism that encrypts all data exchanged is enabled ($SECURE$) and when it is disabled, using insecure gRPC communication channels ($INSECURE$). For the Mobility service, we use the GetSpeed method, which returns the current drone speed as a float, and the GetPosition method, which returns the current latitude, longitude and altitude as floats. For the Camera service, we use the CaptureImage method using a resolution of 1920x1080, which returns just a string with the file name of the image taken, and the RetrieveImage method requesting an image of 1.2MB size, which is typical for this kind of resolution. The results for various service primitives, presented in Figure 8.8a, show that for the GetSpeed, GetPosition and CaptureImage methods, where the service response is small in size, the additional delay when using the SECURE vs INSECURE configuration is negligible. We note that the high invocation latency of the CaptureImage method in the INSECURE configuration is due to the picamera hardware. On the contrary, the latency of the RetrieveImage method increases by about 180% in the SECURE configuration. This is due to the fact that this call returns a large byte array that needs to be encrypted/decrypted.

To further examine the effect of the security configuration to the invocation latency of the camera-related methods when capturing/retrieving images of different qualities, we use the camera setups listed in Table 8.3. The results, presented in Figures 8.8b and 8.8c show that the increase percentage for the CaptureImage does not exceed the 1% in all cases, while for the RetrieveImage it ranges from 140% up to 220%. In particular, for very high image resolutions (8MP), the overhead is over 300ms, resulting in a total invocation time of 450ms that has to be taken into account by the application developer.

AccessControl policies. To grant an application request access to the target service, the Policy Checker performs two actions: it retrieves the current drone position from the autopilot converting it into a form that can be checked against the regions format used by the access control policies (the respective delay is denoted as T_{pos}), and checks if the position lies inside any of the regions of the policies associated with the

TABLE 8.3: Camera setups used in the experiments

Name	Resolution	Typical Size
`LowRes (VGA)`	640x480	180KB
`720p (HD)`	1280x720	490KB
`1080p (FHD)`	1920x1080	1.2MB
`8MP`	3280x2464	4.1MB



(A) Invocation overhead due to AccessControl policies

(B) Invocation latency of RetrieveImage due to PrivacyControl policies (log scale)

FIGURE 8.9: Service access overheads caused by different policies

application (the delay is denoted as T_{check}). We vary the number of AccessControl policies and measure at the *service-side* the worst case introduced latency, i.e., assuming all regions specified by policies are checked. The results, shown in Figure 8.9a, indicate that the introduced overhead mainly comes from T_{pos}, which remains fixed at around 3.7ms. On the other hand, as expected T_{check} increases with the number of policies, however, in absolute values it remains relatively low even when 100 AccessControl policies are considered ($\sim$ 2.5ms). For a more typical scenario with 10 installed policies per application, the total overhead is less than 4ms, which is practically negligible.

Corrective actions. Depending on the method invoked, after granting access to the service call, one or more policy types may need to be checked to determine whether corrective actions should be applied. For instance, in the Goto method of the mobility service the target location has to conform with the geofence policies, and the target altitude has to be within the bounds set by control limitation policies. The respective overheads when one policy of each type is present are 13ms and 2.5ms, resulting in a total invocation delay of 25ms, which has no effect on the application for typical service usage.

Regarding the camera service, we investigate how the different privacy-preserving actions affect the invocation latency of the RetrieveImage method. Specifically, we consider the $BLUR_{AV}$ and $BLUR_{MED}$ actions which apply blurring to the whole image by convolving it with a low-pass filter kernel using the averaging and the median method, respectively, and the $ANON_{PIXEL}$ and $ANON_{BLUR}$ actions which detect faces and anonymize them using pixels of a fixed color or the Gaussian blurring method, respectively. As a reference, we compare with the case where no privacy action is applied (*None*).

The results are shown in Figure 8.9b for different camera resolution settings. As expected, the overhead increases for higher image resolution. For simple actions like $BLUR_{AV}$, the latency is relatively small in absolute terms, growing up to 1.5s for larger 8MP images. For the more fine-grained actions, the most costly being $ANON_{BLUR}$, the latency grows up to 3 seconds, even for the three lowest resolutions, while for the 8MP setup it goes up to 7seconds. Clearly, such latencies show the limits of the RPi and have to be taken into account by the programmer when developing components that invoke such (potentially) time-consuming methods.

Network switching mechanism. We measure the delays associated with the network switching functionality of Fractus for a scenario where the drone interacts via 4G with the instances of hybrid components located in the cloud (default) and enters the WiFi range of an edge node hosting instances of these hybrid components. Once ad hoc WiFi communication is established with the edge node, which takes on average 620 ms, the Fractus net-proxy components start the performance measurements, which can take several seconds depending on the application requirements and the associated metrics. Note that this does not introduce any delay at the application level since the drone components continue to interact with the hybrid components on the cloud via 4G. If it is decided to switch the target instance to the one located at the edge, the time needed to start redirecting application traffic over WiFi can be expressed as $T_{redir} = T_{info} + T_{apply}$, where T_{info} is the time to exchange routing information, which takes about 7.5 ms, and T_{apply} is the time required to set the routing table rules at both sides. T_{apply} can be expressed as $2 \times N \times T_{rule}$, where N is the total number of service-providing components on the drone and edge node (for each one, a rule needs to be set at both sides) and T_{rule} is the time required to set a single routing rule, on average 55 ms. For example, if the edge node provides one service to the drone (N=1), T_{redir} is measured at about 120 ms, which increases linearly with the number of service-providing components, e.g., for N=4 it takes about 450 ms.

8.9.2 Field experiments

Setup. Our field experiments focus on testing the drone-related functionality in real-world conditions. Thus, the Kubernetes cluster includes the drone and the Fractus control plane, which is hosted on a Dell Precision Tower 5810 server running a standard Linux distribution (Ubuntu 18.04) and is connected to the drone via a VPN. Figure 8.10 shows the experiment's overview, with the drone located at HOME. We use a typical application consisting of two drone components: The *Navigator* is a driver with partial control at two points CTRL1 and CTRL2. At CTRL1, it follows path-based navigation visiting in sequence WP1, WP2, WP3 and WP4. At CTRL2, it follows region-based navigation moving inside REGION2. The *AerialViewer* is a passenger component that captures images periodically, every 10 seconds, while the drone is in these areas of interest.

Application deployment and system chauffeur. At first, Fractus calculates the drone operation areas. For CTRL1 the area is the dilated polygon resulting by applying a (configurable) safety radius of 7 meters to the line segment specified via the provided waypoints (CTRL1 → WP1 → WP2 → WP3 → WP4), whereas for CTRL2 the area is the user-defined region (light blue and grey shaded areas respectively). Since there is a single drone candidate, this is selected as host for the deployment of the application components. Furthermore, as the *Navigator* requests only partial control, and the initial drone location (HOME) does not match the first control point (CTRL1), Fractus introduces a chauffeur to arm the drone, take off to a default altitude and move to CTRL1, where the application driver (*Navigator*) takes over. When the drone reaches WP4 and returns control, the chauffeur drives the drone to

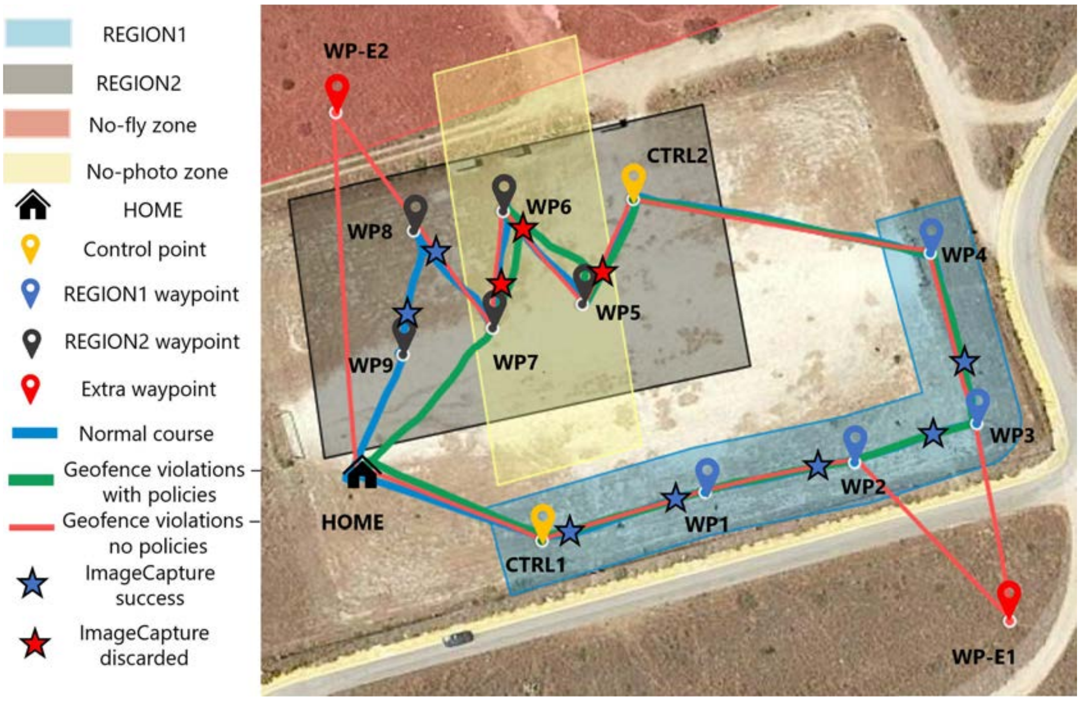


FIGURE 8.10: Overview of the field experiment

the second control point (CTRL2). Inside this region, the *Navigator* makes a zig-zag movement (CTRL2 → WP5 → WP6 → WP7 → WP8 → WP9) and returns control. Then, the chauffeur performs a ReturnToLaunch which takes the drone up to 15m, drives it to HOME and lands it.

Policies generation. In the submitted application description, the *Navigator* is granted access to the Goto and GetControl/ReturnControl methods of the mobility service while the drone is inside the operation area, through corresponding AccessControl policies. Similarly, the *AerialViewer* is granted access to the CaptureImage and RetrieveImage methods of the camera service when the drone is in the specified area. Goto commands are further restricted by GeofenceControl policies that by default discard commands with coordinates outside operation area boundaries (returning a corresponding error code to the component). In addition, the red shaded region in Figure 8.10 is considered a sensitive no-fly zone. If the *Navigator* issues a Goto command to this area, it is considered a malicious behavior, and as a result application access to all service calls is revoked and a ReturnToLaunch command for emergency landing to the HOME location is issued. Also, the allowed flight altitude after takeoff is set from 5 to 15 meters above terrain, enforced via LimitControl policies. Finally, the yellow shaded stripe inside REGION2 is considered a no-photo zone, thus a PrivacyControl policy is added to discard calls to the camera service when the drone is in that region.

Normal execution. The bold blue line in Figure 8.10 depicts the course of the drone during a normal execution of the application, and the blue stars indicate the locations where the *Aerial Viewer* successfully captures and retrieves an image. Also, the blue line in Figure 8.11 plots the recorded flight altitude (relative to the takeoff location) during the normal execution where the *Navigator* keeps the drone steadily at 10 meters. The rise to 15 meters in the last phase before landing is due to the default altitude set by the system chauffeur.

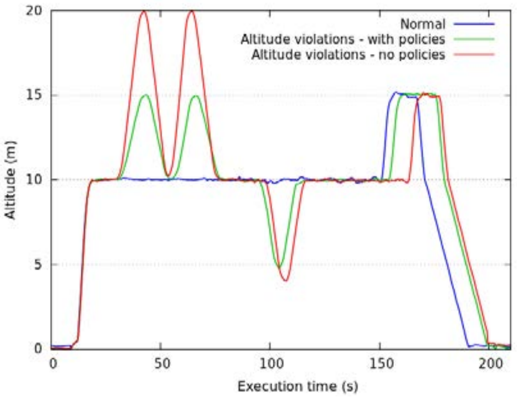


FIGURE 8.11: Drone altitude for normal execution and when the application violates the limits with vs without policies

Enforcement of geofence limits. We program the *Navigator* to deviate from the areas declared in the control points. While the drone is in REGION1, after reaching WP2, we introduce a Goto command to WP-E1, and while the drone is in REGION2, after reaching WP7, we introduce a command to WP-E2 in the no-fly zone. The red trace in Figure 8.10 shows the course followed without any GeofenceControl policies, whereas the green trace shows the course followed with these policies enabled. In the latter case, the drone trace in REGION1 remains unchanged since WP-E1 is simply discarded, whereas in REGION2 the application execution is disrupted after issuing WP-E2, followed by the emergency landing to HOME location under the control of the system.

Enforcement of flight limits. We program the *Navigator* to set the altitude of the Goto commands for WP1 and WP3 to 20 meters, and for WP6 to 4 meters, thereby violating the LimitControl policy for the flight altitude. Figure 8.11 shows the altitude (relative to the takeoff location) during such an execution (green) vs an execution without the altitude LimitControl policy in place (red). As can be seen, every time the application attempts to violate the bounds, Fractus manages to enforce the proper limit.

Enforcement of privacy restrictions. We program the *AerialViewer* to take photos also when the drone is above the no-photo zone inside REGION2 (red stars in Figure 8.10). Such requests are discarded due to the PrivacyControl policy and the respective invocations of the camera service fail, returning an error to the application.

8.9.3 Simulation experiments

Setup. Our simulation experiments focus on illustrating provided functionality, benefits and deployment overhead for more complex scenarios. The setup used, illustrated in Figure 8.12, is based on the simulation environment presented in chapter 2, section 2.3. We configure a multi-node Kubernetes cluster on a single machine using k3d [70], where each node, including drones, is represented by a Docker container with the corresponding Fractus software. The drone's mobility service accesses ArduPilot SITL, the physics simulation is provided by the Gazebo simulator [77] and the camera service accesses Gazebo's virtual camera stream. In edge nodes the camera service returns prerecorded images retrieved from a database. Networking between the Fractus control plane and the nodes is provided through

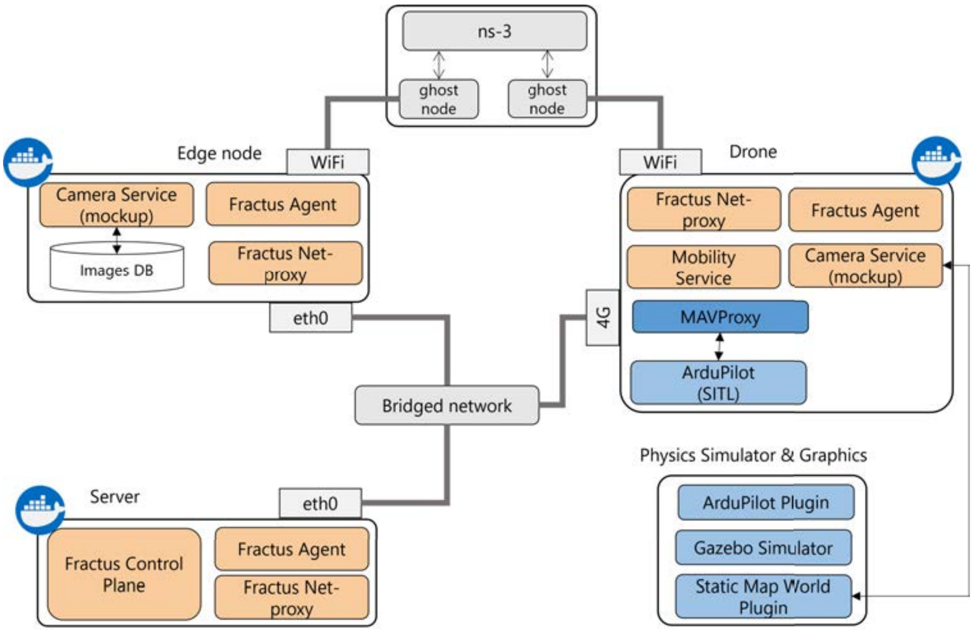
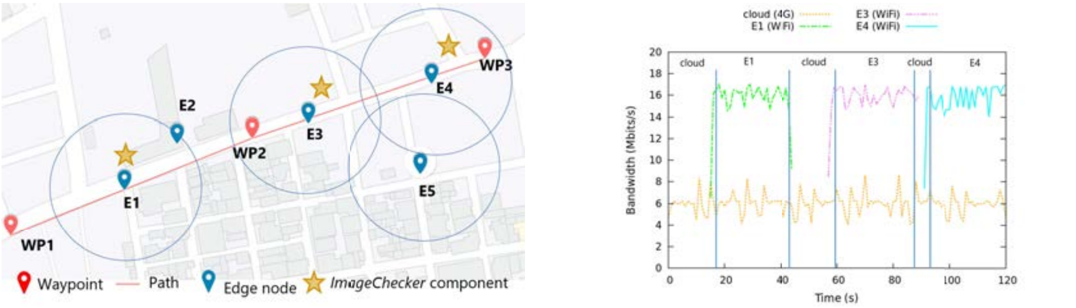


FIGURE 8.12: Simulation setup

an isolated bridge network. For the drone's simulated 4G interface, we set the latency to 60ms and limit the transmission rate to 8Mbps, via the Linux traffic control utility. WiFi networking between drone and edge nodes is through ns-3 [147], set to operate in the ad hoc mode of 802.11g with a data rate of 24Mbps.

Requirements-aware deployment and networking. We use an application with three components mimicking the traffic monitoring application without using the *GroundViewer* and the *Datastore* component. The *Navigator* uses the drone mobility service to take off at WP1, set flying speed to 3m/s, go to WP2 and then to WP3 where it lands the drone. The *AerialViewer* is the drone passenger that retrieves images every 1 second and sends them to a dummy implementation of the hybrid cloud-edge *ImageChecker* that does not perform any actual processing. Target *ImageChecker* selection heuristic is set to maximize the ingress bandwidth. Figure 8.13a shows the application waypoints (red markers), expected path (red line) and the positions of the edge nodes (blue markers); those with a blue circle have WiFi interfaces and the radius indicates their range. The figure also illustrates the deployment of multiple *ImageChecker* instances on the edge nodes (stars). Note that E2 and E5 are filtered-out as the former does not have a WiFi interface and the latter does not offer WiFi coverage in the drone's path.

Figure 8.13b plots the measured bandwidth between *AerialViewer* and the instances of the *ImageChecker* on the cloud and the WiFi-connected edge node as reported by *iperf*. The vertical lines show the intervals where each instance is selected as target. It can be seen that, whenever possible, Fractus favors the nearest instance located at the edge since direct WiFi connectivity offers higher bandwidth. Note that since the drone can have a single direct WiFi link with one of the edge nodes at any point in time, in cases where the WiFi coverage of different edge nodes is overlapping, Fractus switches to the cloud instance as target (via 4G) in order to connect with another edge node that is closer to the drone direction (see 90th second). The intermediate target switch to the cloud instance ensures a continuous data flow at the application level, during the time it takes to establish a direct wireless connection to the next



(A) Waypoints of the application *Navigator* and deployment of the *ImageChecker* on the edge nodes

(B) Measured bandwidth between *AerialViewer* and the *ImageChecker* instances, and selected target instance

FIGURE 8.13: Simulation experiment showcasing the requirements-aware application deployment and networking

edge node and to evaluate the link before deciding to switch the target instance.

Development benefits. We built a version of the above application that accesses the Fractus camera and mobility services and achieves the same behavior without utilizing the network management of Fractus. To this end, we let the *AerialViewer* access directly the drone's WiFi interface to perform the discovery of nearby edge nodes, the connectivity establishment and the traffic redirection. Corresponding code is also placed at the *ImageChecker*. Even for such a simple application, indicative implementations of the components are 193 and 125 lines of code larger compared to their original versions (increases of 240% and 440%, respectively).

Note that, since the management of the network addresses has to be performed manually and the heuristic that leads to network switching is hard coded in the application code, the adaptation for a different system setup or different application needs, requires extra manual programming effort. Also, for the component code to perform these operations, the respective user-level pods must run in privileged mode, which creates many security risks and gives almost unrestricted access to resources on the host system. Further, the deployment of the application components would have to be performed in a hardwired way through manual inspection of each node's networking capabilities. From the above it is clear that Fractus greatly reduces the programming and deployment effort.

Application deployment overhead. The total application deployment time is $T_{deploy} = T_{area} + T_{loc}^{filter} + T_{res}^{filter} + T_{host} + T_{comp}$, consisting of the time needed to calculate the area of operation, filter the candidate hosts based on their location and resource availability, select the best hosts and deploy the application components, respectively. Here, we focus on the overhead of Fractus, leaving out T_{res}^{filter} and T_{comp} which are introduced by Kubernetes. We use the full-fledged traffic monitoring application described in Section 8.2, where the path of the *Navigator* includes 20 waypoints. We take measurements for different configurations where we vary the number of drone and edge nodes while keeping a ratio of 1:5 between them. We also vary their location so that only 10% to 20% (but at least one) is a candidate for hosting each application component.

Figure 8.14 presents the results. At larger scales, most of the overhead comes from T_{loc}^{filter} and in particular the time to filter candidates for the hybrid cloud-edge component. This is due to the number of checks performed when many edge nodes are

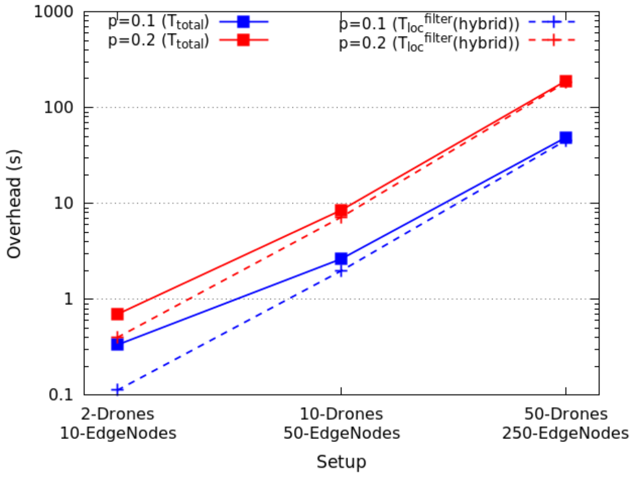


FIGURE 8.14: Deployment overhead of Fractus (y-axis in log scale)

located in the area of operation. One way to reduce this overhead is to stop checking for additional candidates once enough have been found, based on a threshold similar to the *percentageOfNodesToScore* setting in Kubernetes [95].

Chapter 9

Related Work

In this chapter, we discuss previous research related to the orchestration and the systematic testing of drone-based applications. In the first section, we present indicative edge computing platforms that follow different application architecture approaches, drone-specific frameworks and representative approaches for the management of application-level networking. In the second section, we present drone simulators and assessment methods related to cyber-physical systems.

9.1 Edge-related Application Architectures & Orchestration

9.1.1 Edge computing platforms

Edge computing, which places computation and storage at the edge of the network, bringing them closer to mobile devices and sensors, was first introduced as a concept two decades ago, in 2001, under the term cyber foraging [153] with the intend to augment the capabilities of mobile devices by leveraging nearby infrastructure. The establishment however of cloud computing during the mid-2000s favored at first distant data centers for the amplification of mobile resources. This on-demand access to a shared pool of dynamically configured computing resources and to higher-level services, expanded dramatically the capabilities of applications and gave birth to the mobile cloud computing paradigm [36]. Early works, like MAUI [32] and CloneCloud [28], showcased that fine-grained offloading of data processing parts of mobile applications can lead to impressive optimizations in energy consumption and total execution time. Nevertheless, the emergence of mobile applications with high quality of service (QoS) requirements along with large-scale Internet of Things (IoT) deployments made again evident the need for proximity to achieve scalability, responsiveness and privacy preservation. Since then, the interplay between cloud and edge computing gave rise to many models targeting different needs and following different approaches.

Cloudlet [154] proposes a small-scale data center, comprised of a resource-rich computer or cluster of computers, located at the edge of the network for offloading back-end cloud services. In the same vein, ParaDrop [106] and AirBox [14] present edge platforms for deploying functionality on behalf of remote, cloud-based services, with the latter providing edge-related security and privacy by leveraging trusted execution environments. While these approaches try to address bandwidth use and latency requirements in mobile device-cloud communication by introducing a single intermediate layer, CloudPath [119] presents the so-called path computing, a multi-level architecture deployed over the geographic span of the network, from cloud datacenters to the edge, and supports the dynamic hierarchical deployment of cloud services consisting of lightweight, stateless event handlers. In terms of resources provisioning/orchestration, Cloudlet utilizes dynamic VM synthesis and just-in-time provisioning initiated by the mobile devices [56], AirBox and ParaDrop use cloud-based provisioning of Docker containers, and CloudPath provides a Function as a Service (FaaS) system where functionality deployment is based on user-defined preferences and the status of the computing nodes. Our work, in both the platform as a service system and Fractus, presents some similarities with AirBox/ParaDrop regarding the application deployment/resource allocation model and mechanisms. However, on a conceptual level Fractus is not limited on bringing cloud functionality closer to mobile or static IoT devices to serve them, but also provides a transparent computation-communication continuum across drones, edge nodes and the cloud, while addressing key aspects that make drones very different from mobile user devices such as smartphones.

The continuous advances in embedded platforms that provide "deep edge"/leaf devices themselves with more capabilities have created a line of research focused on small local clouds. Two representative systems focused on the orchestration and management of clouds consisting solely of handheld mobile devices are Femtoclouds [57] and Serendipity [160], where the former uses a centralized controller device placed at the edge for the task assignment and scheduling while the latter follows a fully distributed approach. Our work also tries to take advantage of local resources; however, orchestration/scheduling is done in a centralized manner, based on global knowledge from the cloud and known mobility patterns of the drones.

Thematically closer to our work are platforms creating a unified layer of resources that can be accessed transparently by the applications, like PCloud [65] where a personal cloud instance, is formed that seamlessly combines the appropriate resources according to application needs. From a design perspective, our work is more similar with FocusStack [4], which extends the well-known OpenStack platform for managing edge devices as typical datacenters, while we opt for Kubernetes, the most widely adopted container orchestration platform. FocusStack employs a geographical routing layer providing location-based scoping to reduce management communication between edge and cloud and also to enable interactions between edge devices. However, in the presented system prototype, the georouter server through which all these interactions take place is in the cloud, while Fractus supports direct wireless communication between the drone and edge nodes based on their proximity. Moreover, the aforementioned systems are mainly focused to computation and passive sensing applications. Our work goes beyond that point, by including the aspect of mobility control and taking care of the respective requirements.

9.1.2 Drone-specific platforms

There are works targeting solely the orchestration and management of drone applications that follow different approaches. Works like UAV as a Service (UAVaaS) [178] and Dronemap Planner [80] propose service-oriented architectures for virtualizing the access of end-users to multi-tenant drones through web services. These works follow the cloud robotics approach [62] to the extreme, moving all of the application intelligence to powerful infrastructure and making the drones act as simple agents. While this approach can potentially achieve increased resource sharing and prioritization between applications, it eliminates the ability to run application software directly on the drone, which can be quite important for latency-sensitive operations and data-driven application behavior.

Closer to our approach are works that enable the native execution of applications on the drone. For instance, AnDrone [169], proposes a drone as a service (DaaS) solution that combines a cloud service with a virtualization architecture on the drone that enables the execution of container-based applications. However, AnDrone focuses on enabling multiple users to share a drone and run Android-specific applications that access the flight controller and the sensors, rather than on monitoring application behavior and enforcing safety and privacy related rules based on firm agreements. Following a different approach, BeeCluster [61] presents a drone orchestration approach following a predictive optimization strategy for minimizing the total execution time of drone-based sensing tasks. Our work extends these efforts, aiming at the end-to-end deployment of next-generation drone applications that can take full advantage of edge and cloud resources in a transparent way, rather than dealing only with the part of the application that resides on the drone.

Air traffic management & safety. Currently, various efforts are in progress from different official organizations towards low altitude airspace management systems that would enable the integration of drones in urban environments [121, 165, 158]. In the proposed architectures, the core management system is responsible for reviewing flight plans and monitoring those in progress to check their compliance [67]. In case of breaches and emergency situations, drone users/operators are notified to act properly, while collision avoidance is typically handled with sense and avoid technology.

In this thesis, we approach this systems challenge from a software perspective. Unlike the aforementioned systems, we propose a software-based automated solution, which is based on a centralized controller having an up-to-date global view of the running and scheduled operations, and we showcase the ability of enforcing restrictions in case of violations based on pre-flight agreements, without any user involvement. In addition, our approach goes beyond strict airspace management issues and can support a wider range of restrictions through the controlled access of the drone's sensor/actuator payload.

To achieve a degree of safety in drone operations in a non-centralized/ad-hoc manner, the most common practice is to use the geofencing capabilities of the autopilot. This approach can offer increased protection even when piloting the drone via remote control. However, this does not address more refined restrictions, e.g., concerning the use of specific sensors. Also, it does not offer the same level of flexibility for the corrective actions that can be taken in case of a violation, nor the uniformity that can be provided by our approach on top of different autopilot platforms or different versions of the same platform (which may support slightly different features).

Security & isolation. A widely used technique to achieve privacy is to isolate the peripherals and control the access to them, which can be achieved using virtualization (extensions) or hardware security extensions. VirtualDrone [179] utilizes hardware-assisted virtualization and focuses on the attack-resilient control of drones. It creates two separate control environments, one running in a virtual machine where user applications are executed, and a secure one which runs on the host platform. This way, access to peripherals is monitored and in case of violation the application environment is terminated and the secure one takes over. On the other hand, ARM TrustZone [9] trusted hardware components are utilized by PROTC [107] to authorize access to drone peripherals, and AliDrone [108] in order to keep tamper-proof GPS logs and therefore determine the drone's compliance regarding no-fly zones.

Both approaches fit our system architecture and could be adopted as different versions of the drone runtime environment. In our prototype implementations, the desired isolation is based on Docker containers and Kubernetes pods, which are more lightweight and offer the required level of control, given that we do not focus on malicious applications designed with the intention to attack/compromise the software stack of the drone.

9.1.3 Network management

During the last decade a lot of research has been dedicated in the softwarization and virtualization of network functions and systems towards supporting advanced traffic engineering techniques and enabling the dynamic sharing of network resources. Software-defined networking (SDN) proposes an architecture that centralizes the network intelligence by separating the control layer (routing process) from the data layer (actual packet forwarding) [86]. This approach also enables programming the network behavior using software applications, which significantly simplifies network management and operation. In addition, network function virtualization (NFV) virtualizes network functions that were traditionally implemented in hardware equipment (e.g., routers and firewalls) and thus enables the flexible creation of network services. NFV management and orchestration (NFV MANO) frameworks, like OSM [43], follow cloud orchestration techniques for the efficient provisioning of these network services in virtualized infrastructure considering both the Quality of Service (QoS) needs of the different vertical industries being served and the optimal usage of the allocated resources.

The 5G concept adopts this service-oriented view of the network enabled by the combination of SDN and NFV to offer isolated end-to-end virtual networks on top of a single physical network infrastructure and thus realize its vision of satisfying different and possibly contrasting QoS requirements of a variety of applications [181]. While 5G can benefit significantly drone-based applications in various ways, it necessitates extensive deployments by network operators that will take time in order to offer the required wide coverage and stability. Also, most of the static devices at the edge will likely keep using various networking technologies based on specific/targeted application needs [156] [173]. Thus, direct communication with them, if needed, will only be possible through a multi-networking approach like the one we follow in Fractus.

In the edge environment, our work shares similarities with systems that support technology-agnostic interactions. Haggle [163] separates networking details from the application allowing seamless connectivity of web applications across infrastructure and infrastructure-less communication environments. ubiSOAP [23] provides network-agnostic connectivity with QoS-aware network link selection and SOAP-based communication over a multi-network overlay. Similarly, Omni [72] is a more recent effort towards a multi-networking middleware that enables the opportunistic use of wireless technologies by leveraging various device-to-device communication capabilities of IoT devices. Given that ubiSOAP and Omni target opportunistic IoT networking, a large part of these works is dedicated on the discovery of nearby devices offering compatible services. Differently, Fractus uses a centralized cloud layer that has concrete knowledge of the location and available communication technologies of drone and edge nodes and exploits this knowledge to produce suitable component deployment configurations.

In the context of Kubernetes, it is widespread practice to connect application components in a loosely coupled way through the Services abstraction, which defines a logical set of component instances providing the same functionality [89]. Service meshes, like Istio [64], try to completely separate the application's business logic from the communication logic (e.g., service discovery, load balancing, failover) by creating an abstracted application-aware overlay. While these meshes allow fine-grained management through centralized policies, they also introduce extra overhead which is more visible in resource constrained environments due to the injection of sidecar proxy containers in the pods of application components.

Notably, more lightweight, edge-oriented Kubernetes derivatives have been proposed recently, which allow application components to take advantage of various networking technologies. For instance, KubeEdge [87] introduces an MQTT-based communication model through custom protocol mappers but has the drawback that it does not facilitate service-based communication between application components. SMARTER [46] makes use of the device plugin [91] to directly expose the low-level networking interfaces to the application components and provides a very simple container network interface that makes each edge node inaccessible from the others through cloud-style microservices communication. Although these approaches can be definitely useful in certain application fields, they break the familiar service-based interaction pattern between application components and shift the responsibility of connectivity management to the application developer. In contrast, Fractus can transparently take advantage of different ad-hoc networking capabilities under the hood.

9.2 Testing of Drone Applications

9.2.1 Drone simulators

A well-known method for testing drone-based applications before deployment is through software-in-the-loop (SITL) and hardware-in-the-loop (HITL) simulation environments. For instance, [152] describes a SITL platform for the evaluation of control algorithms based on the Microsoft Flight Simulator [116] for the modelling of the flight dynamics and the simulation visualization. HIL-based simulators like [69] and [109] enable the testing of the actual hardware systems. The former assists the validation of drone autopilot hardware and software for a small drone during the development phase, while the latter enables the rapid prototyping of control algorithms specifically for the Piccolo autopilot hardware and supports a wide range of aircrafts. Both configurations employ Matlab/Simulink for the modeling of the drone dynamics and the FlightGear simulator [49] for the simulation visualization.

Another line of work targets the testing of collaborative multi-drone scenarios. MultiUAV [143] is built on Matlab/Simulink and targets the evaluation of cooperative control algorithms for multiple but homogeneous, UAVs. The simulated vehicles include embedded flight software implementing the cooperative control algorithms, vehicle dynamics and a custom autopilot that makes the vehicles capable of waypoint navigation. [59] presents a HITL simulator for multi-aircraft scenarios, which consists of a server that runs flight dynamics model (FDM) instances of JSBsim [175] for each particular aircraft. The drone autopilots interact with the respective FDMs through LabView-specific hardware, which provides the necessary sensor/actuator data forwarding. While these simulation setups share similarities to what we propose, they focus merely on testing the autopilot functionality without simulation support for application/ mission specific scenarios.

Gazebo [77] is a feature-rich 3D robotic simulation platform. It follows a layered architecture that allows high modularity with the creation of new robotic platforms, the addition of new sensors/actuators and the utilization of different physics engines. Also, through suitable plugins it can be used for SITL/HITL drone testing. AirSim [159] is a more recent platform that offers physically and visually realistic simulations through the usage of the Unreal Engine and is focused on enabling developers of autonomous systems to generate large amounts of training data to be used by machine learning algorithms.

Closer to the testing framework that we have developed are platforms that combine different simulation aspects in a single environment. FlyNetSim [12] provides a simulation setup for conducting flexible application experiments using multiple virtual drones with Wi-Fi communication capabilities. Similar to our work, it utilizes ArduPilot SITL [8] for the simulated drones and the ns-3 network simulator [147] for the wireless communication, however each simulated drone corresponds to a thread. While this design decision makes FlyNetSim more lightweight, it does not achieve complete isolation between the different simulation entities. UTSim [2] is another recent simulation framework based on the Unity game engine which is focused on studying air traffic integration issues like sense and avoid, navigation and path planning algorithms. The main drawback is that testing scenarios can be implemented only using its custom user interface, without any integration capabilities regarding the most popular application frameworks in the drone domain, like DroneKit and ROS, and the respective communication protocols, e.g., MAVLink.

In this thesis, we present a modular testing environment based on the AeroLoop system [81], where we significantly enrich the provided functionality by providing a digital twin configuration. Furthermore, we extend its simulation capabilities by integrating more flight dynamics simulators, like Gazebo, offering more networking options and supporting a more lightweight execution through LXDs instead of VMs.

9.2.2 Assessment of cyber-physical systems

There is a variety of approaches towards assessing systems operating in dynamic environments. A broad concept that can be used at all stages of the (design-build-operate) lifecycle of such systems is that of the digital twin (DT), which is a dynamic virtual representation of a physical system/entity, consisting of a simulation model and data coming from the real world. The intended use determines the models to be used. It may vary from achieving better design or manufacturing, to running what-if simulations to predict failures or optimize performance.

Some of the proposed approaches, utilize simulation-based techniques in the development phase to improve the provided functionality. For instance, [58] introduces a simulation-based toolchain during the development of automated driving functions for the identification of critical scenarios in cooperative, automated vehicles. The simulation environment combines a vehicle dynamics simulation of a virtual vehicle, which is the digital twin of the real one, and a traffic simulation, which provides the behavior of the other traffic participants, while the classification process considers standard safety and traffic quality metrics.

Other works employ predictive runtime validation through look-ahead simulation. [17] presents a system that utilizes a simulation-based internal model of a robot in order to accomplish its goal while assuring its safety. To achieve this, it assesses in real-time using the Stage robot simulator [171] all possible actions of the decision search tree (coming from the robot and the dynamic environment), predicts their consequences and selects the most appropriate one. Also, [29] proposes the runtime monitoring of software components through the execution of their digital twin, which are in the form of abstract specifications, in a simulated environment in order to detect and mitigate malicious behaviors.

WCPS [103] and GISOO [5] are cyber-physical systems simulators providing a holistic simulation environment that enables users to evaluate both the control and communication aspects of such systems utilizing the actual embedded code. In both cases, sensor data is generated from a physical model, implemented in Simulink, injected to the corresponding wireless nodes, and later fed back to the controller (also implemented in Simulink), thereby closing the control loop. For the simulation of the wireless nodes WCPS utilizes TOSSIM [102], whereas GISOO employs COOJA [124]. A different approach is followed by CyPhySim [101], a framework for modelling and simulating cyber-physical systems. It is based in Ptolemy II [134] and provides an actor-oriented modelling environment that allows the usage of different models of computation at each level of simulation.

The authors in [22] argue that in software systems offline verification before deployment must be accompanied by quantitative online verification of the key requirements at runtime in order to achieve software dependability and adaptiveness, through the identification and sometimes prediction of requirement violations. Along the same lines, in this thesis, we adopt such a holistic checking approach through a framework that provides the means to test the various software entities of the platform as a service system both in an offline and online fashion.

Chapter 10

Conclusions and Outlook

10.1 Summary

As unmanned autonomous vehicles like drones gain popularity and are being employed in many applications domains, such as surveillance, monitoring and cargo delivery, the need to address the various operation-related challenges is becoming more urgent. In this thesis, we focused on the seamless integration of drones in the cyber-physical landscape by addressing challenges related to the automation of the drone operation cycle, the management of safety and privacy issues and the smooth interaction with existing computing infrastructure. To this end, we have designed and developed system-level mechanisms for strengthening the robustness of drone applications and promoting a more structured, managed and automated application development and deployment approach, properly integrated with established computing paradigms like cloud and edge computing.

An integral part of our research was the systematic experimental evaluation of our work. This was achieved using both a hardware-based testbed consisting of typical computing platforms for drones, embedded nodes and commodity servers, and an integrated emulation/simulation environment, which we developed to support the fast, controlled, safe and flexible testing of more complex scenarios involving multiple communicating virtual versions of the various system entities, which run the same (practically unmodified) software as in real deployments and field tests.

Our main contributions are grouped in two parts.

In the first part of this thesis, we answered the question "How to offer more dependability to drone operations and strengthen their overall automation capability?". In particular, we focused on application scenarios where a centralized mission controller entity is responsible for executing the application logic by coordinating a team of drones via high-level commands. We designed two active replication schemes for tolerating failures critical to the application progress in order to minimize the need for human intervention. Even though the fundamental principles of active replication have been laid out a long time ago, the special features of this particular type of system required a different approach.

At first, we addressed fail-stop failures of the mission controller for both deterministic and non-deterministic applications by utilizing logging and checkpointing of communication and application state information. Then, we addressed Byzantine failures, where the mission controller may behave arbitrarily due to malicious attacks or hardware and software errors, by introducing an agreement protocol. In the latter case, our approach relied on synchronous communication with signed messages, requiring $N = 2 \times f + 1$ replicas to tolerate f failures.

We also discussed concrete implementations of the proposed approaches for an existing programming framework targeting multi-drone applications [85] and identified the key overhead components. In case of deterministic executions, our mechanism for tolerating fail-stop failures introduces overhead only in the event of a drone failure, which depends on the size of the log entries that need to be exchanged between the controller replicas. In case of non-deterministic executions, the overhead is dominated by the transfer time of the application execution state produced during the non-deterministic operations. Finally, the overhead of our Byzantine fault-tolerant mechanism mainly comes from the agreement protocol between the controller replicas.

Given that in the system model used in our work the mission controller is responsible for the high-level coordination of the drones without being involved in tight control loops related to the drone stability and the obstacle avoidance, we conclude that the provided benefits in terms of system robustness can surpass the overhead introduced by our mechanisms. We note however that the selection of the appropriate fault-tolerant mechanism should be application-specific and depend on the trade-off between the respective overhead and the application's criticality level / fault-tolerance needs.

In the second part of this thesis, we targeted the question "How to support the flexible and automated deployment and testing of drone-based applications while ensuring their smooth integration in the cyber-physical infrastructure?". To this end, we designed and implemented two frameworks that can assist application developers and users in the deployment, monitoring and testing processes, as well as the efficient interaction of drones with other system entities in the cloud and at the edge, while ensuring that certain safety and privacy restrictions are not violated. We adopted the platform as a service (PaaS) paradigm of cloud computing, considering drones as another type of resource that can be accessed through a shared infrastructure. However, in our case drones are handled in a special way taking into account their mobility, their onboard sensing/actuation equipment and the fact that they rely on wireless communication.

At first, we presented a holistic approach towards supporting a more reliable managed operation of single-component drone applications through a software platform that takes care of their automated deployment and controlled execution. This platform is coupled with corresponding simulation and digital twin support for detecting bugs before deployment and indicating possible malfunctions during operation in the real world, respectively. Then, we presented Fractus, an orchestration framework targeting distributed, component-based applications that span over the entire system landscape, including drones, static edge nodes and the cloud. Fractus supports the mission-aware placement of the application components and the transparent redirection of application traffic through ephemeral direct communication links. This is achieved through structured descriptions that accompany the application code and allow users to specify placement and communication requirements in an intuitive way, as well as corresponding, extensible, system-level mechanisms. Further, safety and privacy constraints are enforced through the policy-based access of critical resources.

We also discussed concrete implementations of the proposed platforms based on mature software deployment and simulation technologies. Using a real drone in the field and a simulation setup, we demonstrated the provided functionality through indicative scenarios. The functional evaluation of Fractus is accompanied by an extensive quantitative evaluation showing that it decreases the development effort while incurring acceptable overhead.

We view the shift to a more automated application lifecycle management approach and the existence of multiple safety layers as key elements towards the wider adoption of drones in the context of next-generation applications that may incorporate data from various edge and cloud sources. While our contributions are steps in this direction, more challenges lie ahead. For instance, the inherently distributed nature of the devices located at the edge of the network make the proposed systems vulnerable to physical attacks. In addition, despite the emerging interest from industry and the establishment of regulations that will enable safe drone flights even in urban environments, drone- and edge-related systems strive to find the financial incentives that will establish them in the current computing landscape. With the first, pilot automated drone flights taking already place in several countries more typical usage scenarios will likely be standardized in the following years. In turn, this will allow larger scale deployments where the proposed approaches could contribute towards the creation of a lively ecosystem.

10.2 Future Work

There are several open research opportunities and practical issues related to our work that are worth pursuing in the future.

Since the overhead of the fail-stop fault-tolerant mechanism in deterministic and non-deterministic execution depends on the time needed to exchange the logs and checkpoint images, respectively, one could investigate ways to reduce these sizes. For instance, the checkpoint image sizes could be substantially reduced by taking into account the specific properties and features of the TeCoLa runtime environment used in our implementation. While compact representations of logs/checkpoints could reduce the corresponding sizes one should also explore the trade-offs between the compression ratio and the total overhead including the compression delay. In addition, the assessment of different radio technologies for the replica-node communication domains and the evaluation of our implementation in the field, using an appropriate multi-drone testbed, would provide valuable insight of the system operation and the achieved performance under real-world conditions.

Regarding the Byzantine fault tolerance scheme, from a theoretical perspective, one could provide a formal proof of the approach, based on key safety and progress properties. From a practical perspective, since the overhead mainly comes from the synchronization between the replicas, performance could be improved using a faster network in the replica domain and cheaper signature/verification methods. Our work could also serve as a basis for investigating Byzantine fault-tolerant mechanisms for more relaxed assumptions closer to an asynchronous system, as well as for exploring more flexible schemes to allow less-critical parts of the mission program (such as the retrieval of drone state information) to be executed with weaker fault-tolerant properties. Finally, it would be interesting to support the dynamic replacement of faulty replicas without having to suspend/resume the mission.

Several aspects of the PaaS system could be extended. For instance, one might enrich the types of corrective actions available to the user in order to provide more expressiveness regarding the handling of attempted violations. Also, as a next step to this, one could introduce a full-fledged exception handling mechanism that would allow the application program to be aware of the violation attempts and implement its own handlers for them.

There are also different directions that could be followed to improve and extend our offline and runtime testing approach. On the one hand, one might explore ways of enriching the digital twin setup in order to have the ability to run predictive simulations at runtime. On the other hand, it could be of interest to integrate yet another form of runtime testing, through the support of suitable drills that imitate specific problematic situations in the PaaS in order to check the successful triggering of the respective compensating actions.

Finally, there are several ways to extend the functionality and capabilities of Fractus. For example, one could explore the parallel usage of multiple drones by the same application, as well as the transparent hand-over from one drone to another to support longer application missions that go beyond the operational autonomy of a single drone. This, in turn, brings up the need for more advanced drone allocation, application deployment and monitoring/enforcement processes and mechanisms. It is also worth investigating a better isolation between Fractus and the applications running on drones and edge nodes, through the integration of trusted execution environments like ARM TrustZone [9] or more secure container runtimes that provide an extra isolation layer, such as gVisor [55]. Since these techniques incur a performance penalty, such an approach should take into account the security vs performance trade-offs. Further, a more standardized way of describing the application structure and requirements of each component could be achieved, for instance, by extending the OASIS TOSCA [167] specification, to have portability over different deployment technologies.

Bibliography

[1] AeroLoop Project. *Flexible experimentation with virtual UAVs through a software-in-the loop and hardware-in-the-loop simulation infrastructure*. https://aeroloop.e-ce.uth.gr/. 2022.

[2] Amjed Al-Mousa et al. "UTSim: A Framework and Simulator for UAV Air Traffic Integration, Control, and Communication". In: *International Journal of Advanced Robotic Systems* 16.5 (2019). DOI: 10.1177/1729881419870937.

[3] Lorenzo Alvisi and Keith Marzullo. "Message Logging: Pessimistic, Optimistic, Causal, and Optimal". In: *IEEE Transactions on Software Engineering* 24.2 (1998), pp. 149–159. DOI: 10.1109/32.666828.

[4] Brian Amento et al. "FocusStack: Orchestrating Edge Clouds Using Location-Based Focus of Attention". In: *Proc. IEEE/ACM Symposium on Edge Computing (SEC)*. 2016, pp. 179–191. DOI: 10.1109/SEC.2016.22.

[5] Behdad Aminian et al. "GISOO: A Virtual Testbed for Wireless Cyber-Physical Systems". In: *Proc. 39th Annual Conference of the IEEE Industrial Electronics Society (IECON)*. 2013, pp. 5588–5593. DOI: 10.1109/IECON.2013.6700049.

[6] Jason Ansel, Kapil Arya, and Gene Cooperman. "DMTCP: Transparent checkpointing for cluster computations and the desktop". In: *Proc. IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2009. DOI: 10.1109/IPDPS.2009.5161063.

[7] ArduPilot. *Open source autopilot*. http://ardupilot.org. 2022.

[8] ArduPilot. *SITL Simulator*. http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html. 2022.

[9] ARM. *Trustzone*. https://developer.arm.com/ip-products/security-ip/trustzone. 2022.

[10] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quema. "RBFT: Redundant Byzantine Fault Tolerance". In: *Proc. 33rd IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2013, pp. 297–306. DOI: 10.1109/ICDCS.2013.53.

[11] Jean-Philippe Aurambout, Konstantinos Gkoumas, and Biagio Ciuffo. "Last Mile Delivery by Drones: An Estimation of Viable Market Potential and Access to Citizens Across European Cities". In: *European Transport Research Review* 11.1 (2019). DOI: 10.1186/s12544-019-0368-2.

[12] Sabur Baidya, Zoheb Shaikh, and Marco Levorato. "FlyNetSim: An Open Source Synchronized UAV Network Simulator Based on Ns-3 and Ardupilot". In: *Proc. ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. 2018, 37––45. DOI: 10.1145/3242102.3242118.

[13] Nels Beckman and Jonathan Aldrich. "A Programming Model for Failure-Prone, Collaborative Robots". In: *Proc. International Workshop on Software Development and Integration in Robotics (SDIR)*. 2007. URL: http://www.cs.cmu.edu/~claytronics/papers/beckman-sdir07.html.

[14] Ketan Bhardwaj et al. "Fast, Scalable and Secure Onloading of Edge Functions Using AirBox". In: *Proc. IEEE/ACM Symposium on Edge Computing (SEC)*. 2016, pp. 14–27. DOI: 10.1109/SEC.2016.15.

[15] Bharat Bhargava and Shu-Renn Lian. "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - An Optimistic Approach". In: *Proc. 7th IEEE Symposium on Reliable Distributed Systems (RELDIS)*. 1988, pp. 3–12. DOI: 10.1109/RELDIS.1988.25775.

[16] Jan Dyre Bjerknes and Alan F. T. Winfield. "On Fault Tolerance and Scalability of Swarm Robotic Systems". In: *Springer Tracts in Advanced Robotics*. Springer, 2013, pp. 431–444. DOI: 10.1007/978-3-642-32723-0_31.

[17] Christian Blum, Alan F. T. Winfield, and Verena V. Hafner. "Simulation-Based Internal Models for Safer Robots". In: *Frontiers in Robotics and AI* 4 (2018). DOI: 10.3389/frobt.2017.00074.

[18] Flavio Bonomi et al. "Fog Computing: A Platform for Internet of Things and Analytics". In: *Big Data and Internet of Things: A Roadmap for Smart Environments*. Springer International Publishing, 2014, pp. 169–186.

[19] Daniel Browning. *UAVs can play a vital role in the future of smart cities*. Ed. by Smart Cities Dive. https://www.smartcitiesdive.com/news/uavs-can-play-a-vital-role-in-the-future-of-smart-cities/586857/ (2020-10-13).

[20] Navin Budhiraja et al. "The Primary-Backup Approach". In: *Distributed Systems (2nd Ed.)* Pearson, 1993, pp. 199–216. ISBN: 0201624273.

[21] omni calculator. *Drone flight time formula*. https://www.omnicalculator.com/other/drone-flight-time#drone-flight-time-formula. 2022.

[22] Radu Calinescu et al. "Self-Adaptive Software Needs Quantitative Verification at Runtime". In: *Communications of the ACM* 55.9 (2012), pp. 69–77. DOI: 10.1145/2330667.2330686.

[23] Mauro Caporuscio, Pierre-Guillaume Raverdy, and Valerie Issarny. "ubiSOAP: A Service-Oriented Middleware for Ubiquitous Networking". In: *IEEE Transactions on Services Computing* 5.1 (2012), pp. 86–98. DOI: 10.1109/TSC.2010.60.

[24] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *Proc. 3rd Symposium on Operating Systems Design and Implementation (OSDI)*. 1999, pp. 173–186. ISBN: 9781880446393.

[25] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Transactions on Computer Systems* 3.1 (1985), pp. 63–75. DOI: 10.1145/214451.214456.

[26] Jo-Mei Chang and N. F. Maxemchuk. "Reliable Broadcast Protocols". In: *ACM Transactions on Computer Systems* 2.3 (1984), pp. 251–273. DOI: 10.1145/989.357400.

[27] Anders Lyhne Christensen, R. O'Grady, and Marco Dorigo. "From Fireflies to Fault-Tolerant Swarms of Robots". In: *IEEE Transactions on Evolutionary Computation* 13.4 (2009), pp. 754–766. DOI: 10.1109/TEVC.2009.2017516.

[28] Byung-Gon Chun et al. "CloneCloud: Elastic Execution between Mobile Device and Cloud". In: *Proc. 6th Conference on Computing Systems (EuroSys)*. 2011, pp. 301–314. DOI: 10.1145/1966445.1966473.

[29] Emilia Cioroaica et al. "Towards Runtime Monitoring for Malicious Behaviors Detection in Smart Ecosystems". In: *Proc. IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019, pp. 200–203. DOI: 10.1109/ISSREW.2019.00072.

[30] Computer Systems Lab. *Department of Electrical and Computer Engineering, University of Thessaly, Greece*. https://csl.e-ce.uth.gr/. 2022.

[31] George F. Coulouris et al. *Distributed Systems: Concepts and Design (5th Ed.)* Pearson, 2012. ISBN: 9780132143011.

[32] Eduardo Cuervo et al. "MAUI: Making Smartphones Last Longer with Code Offload". In: *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 2010, pp. 49–62. DOI: 10.1145/1814433.1814441.

[33] Yanzhe Cui et al. "ReFrESH: A Self-Adaptation Framework to Support Fault Tolerance in Field Mobile Robots". In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2014, pp. 1576–1582. DOI: 10.1109/IROS.2014.6942765.

[34] Xavier Défago and André Schiper. "Semi-Passive Replication and Lazy Consensus". In: *Journal of Parallel and Distributed Computing* 64.12 (2004), pp. 1380–1398. DOI: 10.1016/j.jpdc.2004.08.006.

[35] Dell. *Dell Precision Tower 5810 spec sheet*. https://i.dell.com/sites/csdocuments/Shared-Content_data-Sheets_Documents/en/al/CSG-EN-XX-ALL-Dell-Precision-Tower-5810-spec-sheet.pdf. 2022.

[36] Hoang T. Dinh et al. "A Survey of Mobile Cloud Computing: Architecture, Applications, and Approaches". In: *Wireless Communications and Mobile Computing* 13.18 (2011), pp. 1587–1611. DOI: 10.1002/wcm.1203.

[37] Danny Dolev and H. Raymond Strong. "Authenticated Algorithms for Byzantine Agreement". In: *SIAM Journal on Computing* 12.4 (1983), pp. 656–666. DOI: 10.1137/0212045.

[38] DroneKit. *Developer tools for drones*. http://dronekit.io/. 2022.

[39] EASA. *Civil drones (Unmanned Aircraft)*. https://www.easa.europa.eu/domains/civil-drones. 2022.

[40] Elizabeth Ciobanu. *How Much Does Drone Insurance Cost?* Ed. by droneblog.com. https://www.droneblog.com/how-much-does-drone-insurance-cost/. Accessed: 2019-01-10. 2021.

[41] E. N. (Mootaz) Elnozahy et al. "A Survey of Rollback-recovery Protocols in Message-passing Systems". In: *ACM Computing Surveys* 34.3 (2002), pp. 375–408. DOI: 10.1145/568522.568525.

[42] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. ISBN: 978-0131858589.

[43] ETSI. *OSM Release FIVE Technical Overview*. Tech. rep. Jan. 2019. URL: https://osm.etsi.org/images/OSM-Whitepaper-TechContent-ReleaseFIVE-FINAL.pdf.

[44] Mark Louis Fairbairn, Iain Bate, and John A. Stankovic. "Improving the Dependability of Sensornets". In: *Proc. IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*. 2013, pp. 274–282. DOI: 10.1109/DCOSS.2013.80.

[45] Pesech Feldman and Silvio Micali. "An Optimal Probabilistic Protocol for Synchronous Byzantine Agreement". In: *SIAM Journal on Computing* 26.4 (1997), pp. 873–933. DOI: 10.1137/S0097539790187084.

[46] Alexandre Ferreira et al. "SMARTER: Experiences with Cloud Native on the Edge". In: *USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*. 2020. URL: https://www.usenix.org/conference/hotedge20/presentation/ferreira.

[47] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *Journal of the ACM* 32.2 (1985), pp. 374–382. DOI: 10.1145/3149.214121.

[48] flannel. *Network fabric for containers, designed for Kubernetes*. https://github.com/flannel-io/flannel. 2022.

[49] FlightGear. *Flight Simulator*. https://www.flightgear.org/.

[50] Open Source Robotics Foundation. *SDF - Simulation Description Format*. http://sdformat.org/. 2022.

[51] Juan A. Garay and Yoram Moses. "Fully Polynomial Byzantine Agreement for Processors in Rounds". In: *SIAM Journal on Computing* 27.1 (1998), pp. 247–290. DOI: 10.1137/S0097539794265232.

[52] GEOS. *Geometry Engine, Open Source*. https://libgeos.org/. 2022.

[53] Rachid Guerraoui and André Schiper. "Fault-tolerance by Replication in Distributed Systems". In: *Proc. Reliable Software Technologies — Ada-Europe '96*. Springer, 1996, pp. 38–57. DOI: 10.1007/BFb0013477.

[54] Pinyao Guo et al. "RoboADS: Anomaly Detection Against Sensor and Actuator Misbehaviors in Mobile Robots". In: *Proc. 48th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 574–585. DOI: 10.1109/DSN.2018.00065.

[55] gVisor. *Application Kernel for Containers*. https://gvisor.dev/. 2022.

[56] Kiryong Ha et al. "Just-in-Time Provisioning for Cyber Foraging". In: *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 2013, pp. 153–166. DOI: 10.1145/2462456.2464451.

[57] K. Habak et al. "Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge". In: *Proc. IEEE International Conference on Cloud Computing (CLOUD)*. 2015, pp. 9–16. DOI: 10.1109/CLOUD.2015.12.

[58] Sven Hallerbach et al. "Simulation-Based Identification of Critical Scenarios for Cooperative and Automated Vehicles". In: *SAE International Journal of Connected and Automated Vehicles* 1.2 (2018), pp. 93–106. DOI: 10.4271/2018-01-1066.

[59] Swaroop A. Hangal, Bharat Tak, and Hemendra Arya. "Distributed Hardware-In-Loop Simulations for multiple Autonomous Aerial Vehicles". In: *Proc. AIAA Modeling and Simulation Technologies Conference*. 2015. DOI: 10.2514/6.2015-0151.

[60] HBICT. *Hash Based Incremental Checkpointing Tool*. http://hbict.sourceforge.net. Accessed: 2019-01-10. 2022.

[61] Songtao He et al. "BeeCluster: Drone Orchestration via Predictive Optimization". In: *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 2020, pp. 299–311. DOI: 10.1145/3386901.3388912.

[62] Guoqiang Hu, Wee Tay, and Yonggang Wen. "Cloud Robotics: Architecture, Challenges and Applications". In: *IEEE Network* 26.3 (2012), pp. 21–28. DOI: 10.1109/MNET.2012.6201212.

[63] Huawei. *4G Dongle E3372*. https://consumer.huawei.com/en/routers/e3372/. 2022.

[64] Istio. *Service mesh*. https://istio.io/. 2021.

[65] Minsung Jang et al. "Personal Clouds: Sharing and Integrating Networked Resources to Enhance End User Experiences". In: *Proc. IEEE Conference on Computer Communications (INFOCOM)*. 2014, pp. 2220–2228. DOI: 10.1109/INFOCOM.2014.6848165.

[66] JGroups. *A Toolkit for Reliable Messaging*. http://jgroups.org. Accessed: 2019-01-10. 2022.

[67] Tao Jiang et al. "Unmanned Aircraft System Traffic Management: Concept of Operation and System Architecture". In: *International Journal of Transportation*

Science and Technology 5.3 (2016), pp. 123–135. DOI: 10.1016/j.ijtst.2017.01.004.

[68] Jonathan Feist. *Drone prices – how much do drones cost?* Ed. by dronerush.com. https://dronerush.com/drone-price-how-much-do-drones-cost-21540/ (2021-08-05). 2021.

[69] Dongwon Jung and Panagiotis Tsiotras. "Modelling and Hardware-in-the-Loop Simulation for a Small Unmanned Aerial Vehicle". In: *Proc. AIAA Infotech at Aerospace Conference and Exhibit*. 2007. DOI: 10.2514/6.2007-2768.

[70] K3D. *Lightweight wrapper to run k3s in docker*. https://k3d.io/. 2022.

[71] K3S. *Lightweight Kubernetes*. https://k3s.io/. 2022.

[72] Tomasz Kalbarczyk and Christine Julien. "Omni: An Application Framework for Seamless Device-to-Device Interaction in the Wild". In: *Proc. 19th International Middleware Conference*. 2018, pp. 161–173. DOI: 10.1145/3274808.3274821.

[73] Pierre Kancir. *Ardupilot Gazebo plugin*. https://github.com/khancyr/ardupilot_gazebo. 2022.

[74] Rüdiger Kapitza et al. "CheapBFT: Resource-efficient Byzantine Fault Tolerance". In: *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*. 2012, pp. 295–308. DOI: 10.1145/2168836.2168866.

[75] Jonathan Katz and Chiu-Yuen Koo. "On Expected Constant-Round Protocols for Byzantine Agreement". In: *Journal of Computer and System Sciences* 75.2 (2009), pp. 91–112. DOI: 10.1016/j.jcss.2008.08.001.

[76] Adil Khadidos, Richard M. Crowder, and Paul H. Chappell. "Exogenous Fault Detection and Recovery for Swarm Robotics". In: *IFAC-PapersOnLine* 48.3 (2015), pp. 2405–2410. DOI: 10.1016/j.ifacol.2015.06.448.

[77] Nathan Koenig and Andrew Howard. "Design and Use Paradigms for Gazebo, an Open-source Multi-robot Simulator". In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2004, pp. 2149–2154. DOI: 10.1109/IROS.2004.1389727.

[78] Fanxin Kong et al. "Cyber-Physical System Checkpointing and Recovery". In: *Proc. ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. 2018, pp. 22–31. DOI: 10.1109/ICCPS.2018.00011.

[79] Ramakrishna Kotla et al. "Zyzzyva: Speculative Byzantine Fault Tolerance". In: *ACM Transactions on Computer Systems* 27.4 (2010), 7:1–7:39. DOI: 10.1145/1658357.1658358.

[80] Anis Koubâa et al. "Dronemap Planner: A Service-Oriented Cloud-Based Management System for the Internet-of-Drones". In: *Ad Hoc Networks* 86 (2019), pp. 46–62. DOI: 10.1016/j.adhoc.2018.09.013.

[81] Manos Koutsoubelias, Nasos Grigoropoulos, and Spyros Lalis. "A Modular Simulation Environment for Multiple UAVs with Virtual WiFi and Sensing Capability". In: *Proc. 2018 IEEE Sensors Applications Symposium (SAS)*. IEEE, 2018. DOI: 10.1109/SAS.2018.8336766.

[82] Manos Koutsoubelias, Nasos Grigoropoulos, and Spyros Lalis. "Virtual Sensor Services for Simulated Mobile Nodes". In: *Proc. 2017 IEEE Sensors Applications Symposium (SAS)*. IEEE, 2017. DOI: 10.1109/SAS.2017.7894115.

[83] Manos Koutsoubelias and Spyros Lalis. "Coordinated Broadcast-Based Request-Reply and Group Management for Tightly-Coupled Wireless System". In: *Proc. 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 2016, pp. 1163–1168. DOI: 10.1109/ICPADS.2016.0153.

[84] Manos Koutsoubelias and Spyros Lalis. “Fault-Tolerance Support for Mobile Robotic Applications”. In: *Proc. 13th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2018. DOI: 10.1109/SIES.2018.8442098.

[85] Manos Koutsoubelias and Spyros Lalis. “TeCoLa: A Programming Framework for Dynamic and Heterogeneous Robotic Teams”. In: *Proc. 13th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous)*. 2016, pp. 115–124. DOI: 10.1145/2994374.2994397.

[86] Diego Kreutz et al. “Software-Defined Networking: A Comprehensive Survey”. In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. DOI: 10.1109/JPROC.2014.2371999.

[87] KubeEdge. *Kubernetes Native Edge Computing Framework*. https://kubeedge.io/en/. 2021.

[88] Kubernetes. *API Server*. https://kubernetes.io/docs/concepts/overview/kubernetes-api/. 2022.

[89] Kubernetes. *Cluster Networking*. https://kubernetes.io/docs/concepts/cluster-administration/networking/. 2021.

[90] Kubernetes. *Custom Resources*. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. 2022.

[91] Kubernetes. *Device Plugin Framework*. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/resource-management/device-plugin.md. 2022.

[92] Kubernetes. *Network Policies*. https://kubernetes.io/docs/concepts/services-networking/network-policies/. 2022.

[93] Kubernetes. *Operator Pattern*. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/. 2022.

[94] Kubernetes. *Production-Grade Container Orchestration*. https://kubernetes.io/. 2022.

[95] Kubernetes. *Scheduler performance tuning*. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduler-perf-tuning/. 2022.

[96] Kubernetes. *Services*. https://kubernetes.io/docs/concepts/services-networking/service/. 2022.

[97] KVM. *Kernel Virtual Machine*. https://www.linux-kvm.org/. 2022.

[98] Leslie Lamport. “The Part-time Parliament”. In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169. DOI: 10.1145/279227.279229.

[99] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563.

[100] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401. DOI: 10.1145/357172.357176.

[101] Edward A. Lee et al. “Modeling and Simulating Cyber-Physical Systems using CyPhySim”. In: *Proc. International Conference on Embedded Software (EMSOFT)*. 2015, pp. 115–124. DOI: 10.1109/EMSOFT.2015.7318266.

[102] Philip Levis et al. “TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications”. In: *Proc. 1st International Conference on Embedded Networked Sensor Systems (SenSys)*. 2003, pp. 126–137. DOI: 10.1145/958491.958506.

[103] Bo Li et al. “Realistic Case Studies of Wireless Structural Control”. In: *Proc. ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. 2013, pp. 179–188. DOI: 10.1145/2502524.2502549.

[104] Linux Foundation. *PREEMPT_RT patch*. https://wiki.linuxfoundation.org/realtime/. 2022.

[105] Barbara Liskov and Robert Scheifler. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs". In: *ACM Transactions on Programming Languages and Systems* 5.3 (1983), pp. 381–404. DOI: 10.1145/2166.357215.

[106] Peng Liu, Dale Willis, and Suman Banerjee. "ParaDrop: Enabling Lightweight Multi-tenancy at the Network's Extreme Edge". In: *Proc. IEEE/ACM Symposium on Edge Computing (SEC)*. 2016, pp. 1–13. DOI: 10.1109/SEC.2016.39.

[107] Renju Liu and Mani Srivastava. "PROTC: PROTeCting Drone's Peripherals Through ARM TrustZone". In: *Proc. Workshop on Micro Aerial Vehicle Networks, Systems, and Applications (DroNet)*. 2017. DOI: 10.1145/3086439.3086443.

[108] Tianyuan Liu et al. "AliDrone: Enabling Trustworthy Proof-of-Alibi for Commercial Drone Compliance". In: *Proc. 38th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2018, pp. 841–852. DOI: 10.1109/ICDCS.2018.00086.

[109] Mariano I. Lizarraga et al. "Simulink Based Hardware-in-the-Loop Simulator for Rapid Prototyping of UAV Control Algorithms". In: *Proc. AIAA Infotech Conference*. 2009. DOI: 10.2514/6.2009-1843.

[110] LXD. *Linux system-level containers*. https://linuxcontainers.org/lxd/.

[111] Markets and Markets. *Unmanned Aerial Vehicle (UAV) Market*. https://www.marketsandmarkets.com/Market-Reports/unmanned-aerial-vehicles-uav-market-662.html. Accessed: 2019-01-10. 2021.

[112] MAVLink. *Drone communication protocol*. https://mavlink.io/en. 2022.

[113] MAVProxy. *UAV ground station software package for MAVLink based systems*. http://ardupilot.github.io/MAVProxy/html/index.html. 2022.

[114] MAVROS. *MAVLink extendable communication node for ROS with proxy for Ground Control Station*. http://wiki.ros.org/mavros. 2022.

[115] MAVSDK. *SDK for MAVLink*. https://www.dronecode.org/sdk/. 2022.

[116] Microsoft. *Flight Simulator*. https://www.flightsimulator.com/.

[117] mongoDB. *NoSQL database*. https://www.mongodb.com/. 2022.

[118] Susan Morrow. *Privacy and Security Issues with Drones*. Ed. by Infosec Resources. https://resources.infosecinstitute.com/privacy-and-security-issues-with-drones/ (2019-04-23).

[119] Seyed Hossein Mortazavi et al. "CloudPath: A Multi-Tier Cloud Computing Framework". In: *Proc. ACM/IEEE Symposium on Edge Computing (SEC)*. 2017, pp. 1–13. DOI: 10.1145/3132211.3134464.

[120] Malek Murisonon. *Drones Will Be Shot Down Until These Misconceptions Are Tackled*. Ed. by Drone Life. https://dronelife.com/2019/03/04/drones-will-be-shot-down-until-these-misconceptions-are-tackled/ (2019-03-04).

[121] NASA. *Unmanned Aircraft System (UAS) Traffic Management (UTM)*. https://utm.arc.nasa.gov/index.shtml.

[122] Bruce Jay Nelson. "Remote Procedure Call". Rep. CMU-CS-81-119. PhD thesis. Department of Computer Science, Carnegie-Mellon University, 1981.

[123] Brian M. Oki and Barbara Liskov. "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems". In: *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC)*. 1988, pp. 8–17. DOI: 10.1145/62546.62549.

[124] Fredrik Osterlind et al. "Cross-Level Sensor Network Simulation with COOJA". In: *Proc. 31st IEEE Conference on Local Computer Networks (LCN)*. 2006, pp. 641–648. DOI: 10.1109/LCN.2006.322172.

[125] Lynne Parker. "ALLIANCE: An Architecture for Fault Tolerant Multirobot Cooperation". In: *IEEE Transactions on Robotics and Automation* 14.2 (1998), pp. 220–240. DOI: 10.1109/70.681242.

[126] Eugenio Pasqua. *The Leading 5G IoT Use Cases*. https://iot-analytics.com/the-leading-5g-iot-use-cases-2019/. Accessed: 2019-04-20. 2019.

[127] Marshall Pease, Robert Shostak, and Leslie Lamport. "Reaching Agreement in the Presence of Faults". In: *Journal of the ACM* 27.2 (1980), pp. 228–234. DOI: 10.1145/322186.322188.

[128] Jérôme Petazzoni. *Build reliable, traceable, distributed systems with ZeroMQ*. https://us.pycon.org/2012/schedule/presentation/260/. 2022.

[129] picamera. *Pure Python interface to the Raspberry Pi camera module*. https://github.com/waveform80/picamera. 2022.

[130] Carlo Pinciroli and Giovanni Beltrame. "Buzz: An extensible programming language for heterogeneous swarm robotics". In: *Proc. 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 3794–3800. DOI: 10.1109/iros.2016.7759558.

[131] Gazebo static map plugin. *Ground plane model with satellite images*. http://gazebosim.org/tutorials?tut=static_map_plugin. 2022.

[132] Antón Román Portabales and Martín López Nores. "Dockemu: Extension of a Scalable Network Simulation Framework based on Docker and NS3 to Cover IoT Scenarios". In: *Proc. International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*. 2018, 175—182. DOI: 10.5220/0006913601750182.

[133] Randy R. Price. *How to estimate the maximum and recommended flight times of a UAS UAV or Drone System*. Tech. rep. 3469. LSU AgCenter, Jan. 2016. URL: https://www.lsuagcenter.com/portals/communications/publications/publications_catalog/crops_livestock/how-to-estimate-the-maximum-and-recommended-flight-times-of-a-uas-uav-or-drone-system.

[134] Claudius Ptolemaeus, ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL: \url{http://ptolemy.org/books/Systems}.

[135] PV-Auto-Scout Project. *Integrated system for the automated inspection of photovoltaic parks using IR-thermography via autonomous aerial vehicles (drones)*. http://www.pvautoscout.com/. 2022.

[136] Pymavlink. *MAVLink protocol C/C++ implementation*. https://github.com/mavlink/c_library_v1. 2022.

[137] Pymavlink. *Python implementation of the MAVLink protocol*. https://github.com/ArduPilot/pymavlink. 2022.

[138] Pyro. *Full-featured Ground Station Application for the ArduPilot Open Source Autopilot Project*. https://ardupilot.org/planner/. 2022.

[139] Pyro. *Python Remote Objects*. https://pyro4.readthedocs.io/en/stable/. 2022.

[140] QGroundControl. *Intuitive and Powerful Ground Control Station for the MAVLink protocol*. http://qgroundcontrol.com/. 2022.

[141] Michael O. Rabin. "Randomized Byzantine Generals". In: *Proc. 24th Annual Symposium on Foundations of Computer Science (SFCS)*. 1983, pp. 403–409. DOI: 10.1109/SFCS.1983.48.

[142] Sampath Rangarajan, Sachin Garg, and Yennun Huang. "Checkpoints-on-Demand with Active Replication". In: *Proc. 7th IEEE Symposium on Reliable Distributed Systems (RELDIS)*. 1988, pp. 75–83. DOI: 10.1109/RELDIS.1998.740477.

[143] Steven Rasmussen et al. “A Multiple UAV Simulation for Researchers”. In: *Proc. Modeling and Simulation Technologies Conference and Exhibit*. 2003, pp. 11–18. DOI: 10.2514/6.2003-5684.

[144] Raspberry Pi. *Camera Module v2*. https://www.raspberrypi.org/products/camera-module-v2/. 2022.

[145] Raspberry Pi 3 Model B. *Third-generation single-board computer*. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. 2022.

[146] RAWFIE Project. *Road-, Air-, and Water- based Future Internet Experimentation*. http://www.rawfie.eu/. 2022.

[147] George F. Riley and Thomas R. Henderson. “The ns-3 Network Simulator”. In: *Modeling and Tools for Network Simulation*. Springer, 2010, pp. 15–34. DOI: 10.1007/978-3-642-12331-3_2.

[148] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. “A Method for Obtaining Digital Signatures and Public-key Cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126. DOI: 10.1145/359340.359342.

[149] ROS. *Robot Operating System*. https://www.ros.org. 2022.

[150] Davide L. Russell. “State Restoration in Systems of Communicating Processes”. In: *IEEE Transactions on Software Engineering* SE-6.2 (1980), pp. 183–194. DOI: 10.1109/TSE.1980.230469.

[151] Erol Şahin. “Swarm Robotics: From Sources of Inspiration to Domains of Application”. In: *Swarm Robotics (SR)*. Springer, 2005, pp. 10–20. DOI: 10.1007/978-3-540-30552-1_2.

[152] Rafael C. B. Sampaio et al. “FVMS: A Novel SiL Approach on the Evaluation of Controllers for Autonomous MAV”. In: *Proc. IEEE Aerospace Conference*. 2013, pp. 1–8. DOI: 10.1109/AERO.2013.6497415.

[153] M. Satyanarayanan. “Pervasive Computing: Vision and Challenges”. In: *IEEE Personal Communications* 8.4 (2001), pp. 10–17. DOI: 10.1109/98.943998.

[154] Mahadev Satyanarayanan et al. “The Case for VM-Based Cloudlets in Mobile Computing”. In: *IEEE Pervasive Computing* 8.4 (2009), pp. 14–23. DOI: 10.1109/MPRV.2009.82.

[155] Fred B. Schneider. “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”. In: *ACM Computing Surveys* 22.4 (1990), pp. 299–319. DOI: 10.1145/98163.98167.

[156] Henning Schulzrinne. *The smart city is more than 5G – Creating a flexible, heterogeneous and programmable IoT infrastructure*. https://smartcity360.eai-conferences.org/2020/speaker/henning-schulzrinne/ (accessed: 2021-04-01). Keynote Speech at EAI SmartCity360 International Convention 2020. 2020.

[157] senseFly. *Drone Case Studies*. https://www.sensefly.com/industries/case-studies. Accessed: 2019-01-10. 2021.

[158] SESAR. *U-Space*. https://www.sesarju.eu/U-space.

[159] Shital Shah et al. “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles”. In: *Field and Service Robotics*. Vol. 5. Springer, 2017. DOI: 10.1007/978-3-319-67361-5_40. eprint: arXiv:1705.05065.

[160] Cong Shi et al. “Serendipity: Enabling Remote Computing among Intermittently Connected Mobile Devices”. In: *Proc. 13th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*. 2012, pp. 145–154. DOI: 10.1145/2248371.2248394.

[161] Russ Smith. *ODE - Open Dynamics Engine*. https://www.ode.org/. 2022.

[162] Yunmok Son et al. "Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors". In: *Proc. 24th USENIX Conference on Security Symposium (SEC)*. 2015, 881—896. ISBN: 978-1-939133-11-3.

[163] Jing Su et al. "Haggle: Seamless Networking for Mobile Applications". In: *UbiComp 2007: Ubiquitous Computing*. Springer, 2007, pp. 391–408. DOI: 10.1007/978-3-540-74853-3_23.

[164] Ogre3D team. *OGRE - Open-source graphics rendering engine*. https://www.ogre3d.org/. 2022.

[165] Thales Group. *Low Altitude Airspace Management (LAAM) platform*. https://www.thalesgroup.com/en/australia/press-release/thales-and-telstra-join-forces-unlock-potential-low-altitude-airspace (2019-01-03).

[166] J. Timmis et al. "An Immune-Inspired Swarm Aggregation Algorithm for Self-Healing Swarm Robotic Systems". In: *Biosystems* 146 (2016), pp. 60–76. DOI: 10.1016/j.biosystems.2016.04.001.

[167] TOSCA. *OASIS Topology and Orchestration Specification for Cloud Applications*. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca (accessed: 2021-04-01). 2022.

[168] V5 nano. *CUAV AutoPilot*. http://doc.cuav.net/flight-controller/v5-autopilot/en/v5-nano.html. 2022.

[169] Alexander Van't Hof and Jason Nieh. "AnDrone: Virtual Drone Computing in the Cloud". In: *Proc. 14th EuroSys Conference*. 2019, 6:1–6:16. DOI: 10.1145/3302424.3303969.

[170] Edwin Vattapparamban et al. "Drones for Smart Cities: Issues in Cybersecurity, Privacy, and Public Safety". In: *Proc. International Wireless Communications and Mobile Computing Conference (IWCMC)*. 2016, pp. 216–221. DOI: 10.1109/IWCMC.2016.7577060.

[171] Richard Vaughan. "Massively Multi-Robot Simulation in Stage". In: *Swarm Intelligence* 2.2-4 (2008), pp. 189–208. DOI: 10.1007/s11721-008-0014-4.

[172] Bas Vergouw et al. "Drone Technology: Types, Payloads, Applications, Frequency Spectrum Issues and Future Developments". In: *Information Technology and Law Series*. T.M.C. Asser Press, 2016, pp. 21–45. DOI: 10.1007/978-94-6265-132-6_2.

[173] Shane Schick (Verizon). *Will 5G replace Wi-Fi?* https://enterprise.verizon.com/resources/articles/s/will-5g-replace-wifi/ (accessed: 2021-04-01). 2021.

[174] Giuliana S. Veronese et al. "Efficient Byzantine Fault-Tolerance". In: *IEEE Transactions on Computers* 62.1 (2013), pp. 16–30. DOI: 10.1109/TC.2011.221.

[175] Tomáš Vogeltanz and Roman Jašek. "JSBSim library for flight dynamics modelling of a mini-UAV". In: *AIP Conference Proceedings* 1648.1 (2015), p. 550015. DOI: 10.1063/1.4912770.

[176] M. Wiesmann et al. "Understanding Replication in Databases and Distributed Systems". In: *Proc. 20th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2000, pp. 464–474. DOI: 10.1109/ICDCS.2000.840959.

[177] David Wright and Rachel Finn. "Making Drones More Acceptable with Privacy Impact Assessments". In: *The Future of Drone Use*. Vol. 27. T.M.C. Asser Press, 2016, pp. 325–351. DOI: 10.1007/978-94-6265-132-6_17.

[178] Justin Yapp, Remzi Seker, and Radu Babiceanu. "UAV as a Service: Enabling On-demand Access and on-the-fly Re-tasking of Multi-tenant UAVs Using Cloud Services". In: *Proc. IEEE/AIAA Digital Avionics Systems Conference (DASC)*. 2016. DOI: 10.1109/DASC.2016.7778007.

[179] Man-Ki Yoon et al. "VirtualDrone: Virtual Sensing, Actuation, and Communication for Attack-resilient Unmanned Aerial Systems". In: *Proc. 8th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. 2017, pp. 143–154. DOI: `10.1145/3055004.3055010`.

[180] ZeroMQ. *Open-source messaging library*. `https://zeromq.org/`. 2022.

[181] H. Zhang et al. "Network Slicing Based 5G and Future Mobile Networks: Mobility, Resource Management, and Challenges". In: *IEEE Communications Magazine* 55.8 (2017), pp. 138–145. DOI: `10.1109/MCOM.2017.1600940`.

[182] Long Zhang et al. "A Survey on 5G Millimeter Wave Communications for UAV-Assisted Wireless Networks". In: *IEEE Access* 7 (2019), pp. 117460–117504. DOI: `10.1109/ACCESS.2019.2929241`.