Programming and Testing Support for Drone Based Applications

Manos Koutsoubelias

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

Department of Electrical and Computer Engineering
University of Thessaly
Greece

September 2018

Abstract

Programming and Testing Support for Drone Based Applications

by

Manos Koutsoubelias

The evolution and the widespread adoption of low cost and versatile manufacturing technologies such as 3D printing along with the recent software and hardware advances in embedded computing, wireless communications and control systems have sparked a development boom in the field of unmanned vehicles (UVs) for civilian use. Nowadays, UVs which are commonly known as drones are widely available, come at low cost, and are capable of operating largely autonomously in aerial, ground and aquatic environments with limited human intervention. Besides their considerable and continuously improving mobility and navigation properties, drones can carry a diverse set of application-specific payloads, including various types of sensors/actuators as well as powerful computing and communication equipment.

Thanks to these capabilities, drones will become key components of cyber-physical systems. They will revolutionize existing applications, and may very well give birth to entirely new ones. Recently, drone platforms have been adopted in several civilian application domains, spanning from smart agriculture and structural health monitoring to the entertainment industry. However, their full potential has not been unleashed yet. Despite the significant work that has been done by the scientific community over the last decades in various aspects of wireless sensor networks, robotics and distributed systems, application development for drones remains a hard task, especially when multiple drones are involved and need to perform a joint mission in a coordinated manner.

In this thesis we focus on supporting mission program development for unmanned vehicles by reducing the amount and complexity of the programming that is required in order to deal with the problems of communication, coordination, heterogeneity and dynamics in multi-drone missions. We introduce a programming model that provides suitable coordination abstractions and mechanisms regarding resource heterogeneity and dynamics, allowing mission programs to utilize multiple drones in a flexible way. We adopt a centralized service-oriented approach whereby a distinguished entity which is called mission controller runs the mission program logic, which retrieves information from the drones and issues high-level control commands to them. In addition, our programming model provides unified abstractions for controlling individual drones as well as teams of drones with minimal development effort. The model along with its supportive runtime environment comes with an integrated fault tolerant mechanism, which combines checkpointing and logging in order to allow seamless recovery from mission controller failures.

We support our programming model with an efficient coordinated transport protocol that performs reliably 1-N request/reply interactions within a group of processes. The protocol utilizes the broadcast nature of the wireless medium to avoid collisions, while allowing a high rate of interactions between the coordinator process (mission controller) and the ordinary processes (UVs) with low and predictable latency. It also comes with integrated fault handling and group management. The protocol is used as a transport for the commands sent by the coordinator process to the ordinary processes. The same protocol can be used to let the coordinator retrieve state/sensor updates from the ordinary processes via polling. We have also done work to support a dynamic adjustment of the polling rate, based on a method that automatically infers the actual data rate of the ordinary processes.

Last but not least, we have developed a modular simulation infrastructure to perform tests with multiple virtual UVs (vUVs) in a flexible, controlled and safe way. Each vUV is a separate virtual machine, which runs the entire software stack including autopilot coupled to a physics model that simulates the mobility-related behavior of the UV. The system currently supports multi-copters with a virtual camera sensor, which can communicate with each other over a simulated WiFi network. We have successfully used the simulation environment to test various functional aspects of our system software stack, which would be practically impossible or prohibitively expensive to do using real drones.

Περίληψη

Υποστήριξη προγραμματισμού και δοκιμαστικής λειτουργίας εφαρμογών με μη επανδρωμένα οχήματα

Μάνος Κουτσουμπέλιας

Οι πρόσφατες εξελίξεις στους τομείς των ενσωματωμένων συστημάτων, των ασύρματων επικοινωνιών και των συστημάτων αυτομάτου ελέγχου έχουν οδηγήσει στην ραγδαία ανάπτυξη των αυτόνομων μη επανδρωμένων οχημάτων (UVs). Σήμερα τα UVs είναι ευρέως διαθέσιμα στην αγορά σε διάφορες εκδόσεις και τύπους ενώ διαθέτουν δυνατότητες αυτόνομης πλοήγησης στο έδαφος, στον αέρα αλλά και στη θάλασσα. Επίσης, αυτά τα οχήματα μπορεί να εξοπλιστούν με εξειδικευμένους αισθητήρες, ενεργοποιητές και μηχανισμούς αλληλεπίδρασης με το περιβάλλον, ανάλογα με τον σκοπό της εκάστοτε αποστολής/εφαρμογής.

Παρά τον μεγάλο αριθμό και την τεχνολογική ποικιλία σε διαθέσιμα UVs οι εφαρμογές που ωφελούνται από αυτά, όπως για παράδειγμα εφαρμογές από τον τομέα της γεωργίας ακριβείας και της επιτήρησης δασικών περιοχών, κάνουν χρήση συνήθως μεμονωμένων οχημάτων ενός συγκεκριμένου τύπου. Οι περισσότερες από αυτές τις εφαρμογές θα ωφελούνταν πολλαπλάσια αν μπορούσαν να χρησιμοποιηθούν πολλά και ετερογενή UVs. Για παράδειγμα, αυτό θα επέτρεπε να καλυφθούν μεγαλύτερες εκτάσεις και περισσότερα σημεία ενδιαφέροντος σε μικρότερο χρονικό διάστημα. Επίσης η συνδυασμένη χρήση αυτόνομων οχημάτων με συμπληρωματικές δυνατότητες κίνησης, αίσθησης και αλληλεπίδρασης με το περιβάλλον θα δημιουργούσε νέες δυνατότητες στην επιτήρηση αλλά και τον έλεγχο φυσικών διαδικασιών.

Όμως, η χρήση πολλών UVs ταυτόχρονα θέτει νέες προκλήσεις. Ένα καίριο ζήτημα είναι το πώς μπορεί να υποστηριχθεί με όσο γίνεται πιο απλό και ευέλικτο τρόπο ο συντονισμένος έλεγχος μιας ολόκληρης ομάδας από διαφορετικά UVs, μέσα από ένα πρόγραμμα-αποστολή. Ο προγραμματισμός UVs, ιδίως για εφαρμογές που θέλουν να εκμεταλλευτούν πολλά UVs ταυτόχρονα, βρίσκεται ακόμα στα πρώτα στάδια της εξέλιξης.

Η παρούσα διδακτορική διατριβή εστιάζει στην παροχή προγραμματιστικής υποστήριξης για την ανάπτυξη εφαρμογών-αποστολών με πολλαπλά και ετερογενή UVs απαλλάσσοντας τον προγραμματιστή από την διαχείριση της επικοινωνίας, της ετερογένειας, της δυναμικής αλλά και της διαχείρισης βλαβών.

Συγκεκριμένα εισάγουμε ένα προγραμματιστικό μοντέλο που παρέχει κατάλληλες προγραμματιστικές αφαιρέσεις αλλά και μηχανισμούς συντονισμού που αφορούν την ετερογένεια και τη δυναμική των οχημάτων, επιτρέποντας στα προγράμματα των αποστολών να χρησιμοποιούν πολλαπλά UVs με ευέλικτο τρόπο. Υιοθετούμε μια κεντρικοποιημένη υπηρεσιοστρεφή προσέγγιση όπου μια διακεκριμένη οντότητα που ονομάζεται ελεγκτής αποστολής εκτελεί τη λογική του προγράμματος αποστολής, ανακτώντας πληροφορίες από τα UVs και εκτελώντας εντολές ελέγχου υψηλού επιπέδου σε αυτά. Επίσης, το μοντέλο μας παρέχει ενοποιημένες προγραμματιστικές αφαιρέσεις για τον έλεγχο μεμονωμένων UV καθώς και ομάδες από αυτά με ενιαίο τρόπο. Το μοντέλο μαζί με το υποστηρικτικό περιβάλλον εκτέλεσης διαθέτει επίσης ενσωματωμένο μηχανισμό ανοχής βλαβών, ο οποίος συνδυάζει τεχνικές καταγραφής κατάστασης στον ελεγκτή και καταγραφής εντολών στα UVs

έτσι ώστε να επιτρέπεται η απρόσκοπτη αποκατάσταση του προγράμματος αποστολής μετά από βλάβες στον ελεγκτή αποστολής.

Για την υποστήριξη του παραπάνω προγραμματιστικού μοντέλου σχεδιάστηκε και αναπτύχθηκε ένα αξιόπιστο πρωτόκολλο επικοινωνίας που υλοποιεί αποτελεσματικά το μοντέλο επικοινωνίας αίτησης-απάντησης με συντονιστή (coordinated request-reply) για μια ομάδα κόμβων. Το πρωτόκολλο είναι σχεδιασμένο να εκμεταλλεύεται τις ιδιότητες του ασύρματου μέσου αποφεύγοντας τις συγκρούσεις στα εκπεμπόμενα πακέτα καθώς επίσης έχει ενσωματωμένη λογική για τη διαχείριση βλαβών αλλά και της ομάδας των κόμβων. Το πρωτόκολλο χρησιμοποιείται σαν πρωτόκολλο μεταφοράς των εντολών που προέρχονται από τον ελεγκτή προς τους κόμβους και των δεδομένων από τους κόμβους προς τον ελεγκτή μέσω διαδοχικών αιτήσεων ανάκτησης δεδομένων (polling) με συγκεκριμένο ρυθμό αποστολής. Επιπρόσθετα, έχουμε διερευνήσει την δυναμική προσαρμογή αποστολής των αιτήσεων ανάκτησης δεδομένων με βάση την αυτοματοποιημένη εκτίμηση του ρυθμού αποστολής στους κόμβους.

Τέλος, σχεδιάστηκε και αναπτύχθηκε ένα περιβάλλον προσομοίωσης (simulator) για την διεξαγωγή εικονικών αποστολών με UVs που προσομοιώνονται σε λογισμικό. Βασικό πλεονέκτημα σε σχέση με δοκιμές με πραγματικά οχήματα, είναι η δυνατότητα διεξαγωγής ενός μεγάλου εύρους απόλυτα ελεγχόμενων αποστολών και σεναρίων δυσλειτουργίας με πρακτικά μηδενικό κόστος και χωρίς κανένα θέμα ασφάλειας. Το περιβάλλον προσομοίωσης συνδυάζει υπάρχουσες πλατφόρμες προσομοίωσης. Τα εικονικά UVs είναι υλοποιημένα σε εικονικές μηχανές (Virtual Machines) που εκτελούν την πλήρη στοίβα λογισμικού ενός UV συμπεριλαμβανομένου του λογισμικού αυτόματου πιλότου το οποίο είναι διασυνδεδεμένο με το λογισμικό εξομοίωσης της κίνησης του UV. Αυτή τη στιγμή το σύστημα υποστηρίζει αποστολές με σμήνη/ομάδες από πολλά εικονικά πολυκόπτερα που είναι εξοπλισμένα με μία εικονική κάμερα. Τα εικονικά πολυκόπτερα μπορούν να επικοινωνούν μεταξύ τους μέσω ενός εξομοιωμένου ασύρματου δικτύου WiFi.

To Nikos, Antigoni, Haris and Anita.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost I would like to express my gratitude to Professor Spyros Lalis from the University of Thessaly for his excellent supervision and invaluable support throughout these years. Without his guidance and constructive criticism this thesis would not have been possible.

I would also like to thank my old and new colleagues who provided me with enjoyable moments and fruitful discussions all these years. Specifically, I would like to thank Dr. Dimitris Syrivelis, Dr. Giorgis Georgakoudis, Dr. Nikos Tziritas and Nasos Grigoropoulos with whom I shared my research career so far.

Last but not least, I owe many thanks to my family, Nikos, Antigoni and Haris for their patience and unconditional love throughout these years. Also, I would like to thank my soulmate Anita Lefaki, who has been by my side during the hard last miles of this work. The least I can do is dedicate this thesis to them.

Related publications

1. M. Koutsoubelias and S. Lalis. Fault-tolerance support for mobile robotic applications. In *13th International Symposium on Industrial Embedded Systems*, pages 1–10, 2018.

2. M. Koutsoubelias, A. Argyriou, and S. Lalis. Scalable and adaptive polling protocol for concurrent wireless sensor data flows. In *International Conference on Pervasive Computing and Communications Workshops*, 2018.

3. M. Koutsoubelias, N. Grigoropoulos, and S. Lalis. A modular simulation environment for multiple uavs with virtual wifi and sensing capability. In *Sensors Applications Symposium*, pages 1–6, 2018.

4. M. Koutsoubelias, N. Grigoropoulos, and S. Lalis. Virtual Sensor Services for Simulated Mobile Nodes. In *Sensors Applications Symposium*, pages 1–6, 2017.

5. M. Koutsoubelias and S. Lalis. Tecola: A programming framework for dynamic and heterogeneous robotic teams. In *13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 115–124, 2016.

6. M. Koutsoubelias and S. Lalis. Coordinated broadcast-based request-reply and group management for tightly-coupled wireless systems. In *22nd International Conference on Parallel and Distributed Systems*, pages 1163–1168, 2016.

Chapter 1

Introduction

1.1 Motivation

The recent advances in embedded computing, wireless communications and control systems have lead to truly radical developments in the area of unmanned vehicles (UVs). Technology that was –just a few years ago– used exclusively for military purposes, has now become accessible to a larger audience, with the potential not only to revolutionize existing applications, but also to pave the way towards completely new ones.

Different UV platforms are emerging at a fast pace, which can operate on the ground (UGVs), on the water surface (USVs), underwater (UUVs), and in the air (UAVs). In particular, UAVs are gradually being adopted in several application domains, such as agriculture, surveillance and coverage of big events in the sport and entertainment industry [99,103,107]. Furthermore, cheap quadcopters are now fully commoditized and can be found next to game consoles and remote controlled models on the shelves of electronics and toy stores all over the world. As was recently pointed out, drones could very well be the next smartphones [1].

Clearly, UVs will play a key role in next-generation applications. A major challenge in this respect is how to minimize the amount of human involvement. Our vision is for UVs to be controlled by a computer program instead of a human operator, at least in non-critical applications. Indeed, there is great progress in auto-pilot programs that drive the motors and steering elements of a UV so that it can stay at a given position or move between positions in a stable way [40]. Many algorithms also have been developed for simultaneous localization and mapping (SLAM) and obstacle avoidance [46, 56]. Moreover, a lot of work has been done in the area of path-planning to cover a terrain, task-partitioning among multiple UVs, and the formation of suitable ad-hoc networks that support the communication between them [30, 32, 57]. However, the development of computer programs that employ UVs in order to perform not only sensing but also actuation tasks –especially when this involves the simultaneous usage of multiple UVs– remains a hard task, compared to how easy it is to write conventional distributed programs for which a broad selection of mature middleware and transport protocols is available to the developer.

1.2 Focus and Contributions

Driven by the above issue, this thesis addresses the following question: *How to support application development for unmanned vehicles in a structured way, while reducing the complexity that is required to deal with the communication, coordination, heterogeneity, dynamics and failures in such applications?* To achieve this goal, we have done work in three main dimensions: (i) a programming model for writing computer applications that use multiple UVs; (ii) a transport-level protocol that can efficiently support the underlying communication of the programming model; (iii) a simulation environment for testing the entire software stack. These are combined together into an integrated platform for developing and testing applications that employ multiple unmanned vehicles (UVs). Below we give an brief overview, connecting the dots together.

Programming Model

The programming model allows the developer to write applications that engage multiple and possibly heterogeneous UVs in a straightforward and flexible way. We consider UVs that can navigate/operate autonomously, but still need to be coordinated at a higher level in order to perform a joint mission in an orchestrated way. We adopt a centralized approach, whereby a distinguished entity runs the mission program, collects information from all UVs and issues high-level control commands to them, through structured remote service interfaces. The UVs process incoming requests and send back corresponding replies. The mission program can form teams and control them (almost) as if they were a single UV. Appropriate hooks are provided for handling the dynamic addition/removal of UVs during the course of a mission. Moreover, failures during the execution of the mission program are tolerated through a hybrid checkpointing and logging scheme, allowing the mission to be resumed in a largely transparent way.

Transport Protocol

In order to support the above programming model, we have developed a suitable transport protocol for performing 1-N request/reply interactions between a coordinator process (the mission control program) and several ordinary processes (the UVs). The protocol is designed to avoid contention among the communicating processes, practically eliminating collisions and back-off effects during wireless transmission. This leads to a high degree of reliability with practically zero retransmisisons. In turn, this reduces the end-to-end delay of the request/reply interactions, and makes it possible to sustain a very high rate of communication, close to channel capacity. The protocol comes with integrated support for the addition and removal of processes in order to manage a process group. It also deals with process failures, and automatically elects a new coordinator if needed. The same protocol can be used to let the coordinator retrieve state/sensor updates from the ordinary processes via polling. We

have also done work to support a dynamic adjustment of the polling rate, based on a method that automatically infers the actual data rate of the ordinary processes.

Simulation Environment

Performing tests with real UVs requires substantial equipment and human resources, needs a lot of preparation, is time-consuming, raises several safety issues, and can be quite costly in case of system malfunctions and failures that occur due to hidden bugs. These problems are amplified for applications that involve several UVs. To address this issue, we have developed a simulation environment for testing both system-software and mission-software in a flexible, controlled and safe way, so that this can reach a high level of maturity before attempting trials in the field. The user may employ multiple virtual UVs (vUVs) depending on the mission. Each vUV is a separate virtual machine, which hosts the entire software stack as this would run in a real UV, coupled to a physics model that simulates the mobility-related behavior of the UV as a function of platform characteristics, terrain conditions and the control commands received from the local autopilot. The system currently supports quadcopters with a virtual camera sensor, which can communicate with each other over a simulated ad-hoc WiFi network. Importantly, the system is extensible. It is possible to introduce different types of UVs, each with its own physics/mobility model and autopilot, as well as non-UV entities such as a control/ground station that communicates with the UVs over separate network links that can be configured to be more reliable but also with a larger latency and less bandwidth compared to WiFi. One may also include additional virtual sensors, either on-board the UVs or ground-based, to support a wider range of applications.

1.3 Outline

The rest of the thesis is structured as follows. Chapter 2 discusses indicative mission scenarios and gives an overview of the system architecture we adopt in our work. Chapter 3 introduces our programming model for writing missions, and the corresponding runtime support we have developed for it. Chapter 4 discusses the fault-tolerance support we introduced in order to be able to resume and continue a mission after a failure of the mission program. Chapter 5 discusses the underlying transport protocol that is used by our runtime support in order to support the remote interactions of the programming model that take place over the network, in an efficient and reliable way. Chapter 6 discusses a method for dynamic adaptation of the polling rate for the general case where the transport protocol is used to support N-1 data flows, from multiple sensor processes to a single collector process. Chapter 7 discusses the simulation environment we have developed to enable tests with several virtual UVs, which was also used to test our own programming model and runtime support for different non-trivial mission scenarios, without having to use real UVs. Finally, Chapter 8 concludes the thesis and points to some directions for future work.

Chapter 2

Overview

This chapter provides an overview of our approach. To illustrate our vision, we start by presenting indicative examples of missions that employ multiple heterogeneous UVs, and highlight their most important characteristics without entering into the application-specific details. Then, we present the system architecture that is adopted as a baseline for our work, and discuss how we structure our programming and testing support for UV-based applications in light of this architecture.

2.1 Mission Scenarios

Existing applications typically use one UVs or several identical UVs in predefined configurations. Going a step further, we envision more advanced missions that involve several UVs with different computing, sensor and actuator resources, different mobility capabilities, which can be added to and removed from the mission in a dynamic way.

Figure 2.1 illustrates a scenario where a number of unmanned aerial vehicles (drones) is sent out to scan a crop field. Some drones are equipped with infrared and visible light cameras, while others feature sensors that are suitable for air quality monitoring. The data collected from these airborne sensors is processed to detect crop deficiencies and take corrective action; for instance, to add extra fertilizer or spray some pesticide in the areas that are problematic. Since the corresponding machinery can be quite heavy or even dangerous to operate from the air, the addition of extra fertilizer or the spraying of the pesticide is performed using a ground vehicle (rover) with the necessary equipment. The rover is parked on site, and by default remains inactive (bottom-left corner of Figure 2.1a). If the drones detect a problem (Figure 2.1b), the rover is activated, and is instructed to move to the area of interest in order to perform the necessary action (Figure 2.1c).

As an example with even richer dynamics and collaboration options, Figure 2.2 shows a scenario where the objective is to detect and extinguish fires in a forest area. During periods of low humidity and high air temperature where the probability of a fire outbreak is large, the forest is monitored by a group of drones. These are equipped with infrared and visible
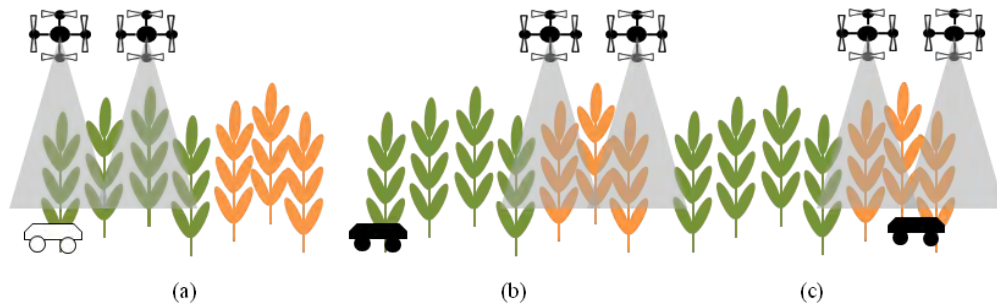
Figure 2.1: Smart agriculture scenario.
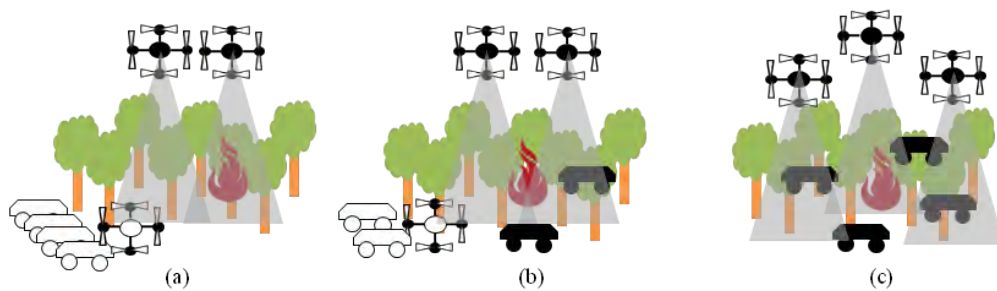
Figure 2.2: Forest surveillance and fire-fighting scenario.

light cameras, and scan the forest while maintaining a suitable formation that maximizes coverage. At a designated area in the forest there is a station where more drones and rovers equipped with firefighting equipment are parked; these remain inactive as long as no fire incident is reported. When the drones detect a fire (Figure 2.2a), the rovers are activated and instructed to move to different positions in order to fight the fire in a coordinated way (Figure 2.2b). If the situation remains critical and the fire could get out of control, additional drones and rovers can join the operation in order to boost the system firefighting capacity (Figure 2.2c). In a similar vein, some of the employed drones and rovers could be withdrawn once the fire is contained.

In a nutshell, the key properties of the missions we target in our work, are as follows:

- **Multiple and potentially heterogeneous UVs.** The mission employs several UVs. The UVs need not all be identical to each other. In fact, UVs can have very different sensing, actuation and mobility capabilities.

- **High-level coordination.** The employed UVs are to a large degree autonomous. As a consequence, the UVs can perform basic (low-level) operations, like physical movement or obstacle avoidance, without any external control by the mission program. The task of the mission program is to coordinate/orchestrate the individual UVs at a higher level, exploiting their specific properties/capabilities in order achieve the mission goals.
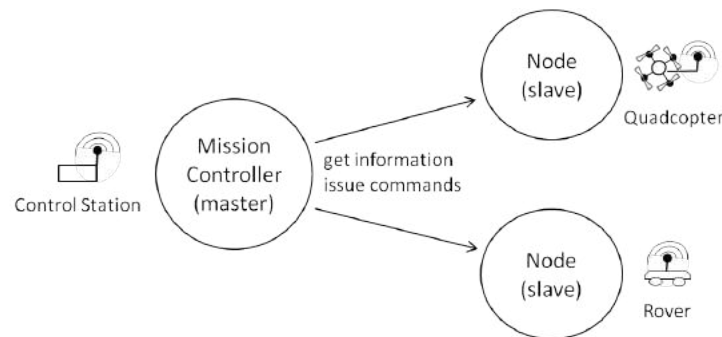
Figure 2.3: Coordination of multiple UVs using a master-slave approach.

- **Loose synchronization.** The desired coordination of the individual UVs and the sensing and actuation tasks to be performed as part of the mission, does not have any tight real-time constraints and can be achieved with infrequent communication.

Our goal is to support the development of UV-based missions with these characteristics.

2.2 Baseline Architecture

As a baseline for our work, we opt for a centralized coordination model, illustrated in Figure 2.3. This follows a master-slave approach, in the spirit of a team leader who monitors all other team members and decides which actions to take next.

More specifically, we model UVs as mobile *nodes* that can provide various types of information via corresponding sensors, and can also perform certain actions via corresponding actuators. Nodes are controlled by a distinguished entity that runs the mission logic. We refer to this entity as the *mission controller*; in a typical deployment, this would be a command/control station. The mission controller acts as a master, which gathers the information that is acquired by the nodes, takes control decisions, and instructs the nodes to perform operations, according to the mission objectives. The nodes merely act as slaves, following the commands of the master.

The centralized approach gives the developer the convenient illusion of programming a single platform, and simplifies the task of writing code that coordinates the individual nodes as one does not have to think in a distributed way. Moreover, the software that runs on the nodes is simplified and can be structured in the form of distinct services, which also makes it possible to re-use and combine nodes in a flexible way to support different application scenarios.

A fundamental drawback of centralized designs is that they have a single point of failure. In our case, the single point of failure is the mission controller. If it crashes during the mission, the nodes will be left without a master. We address this problem by introducing suitable fault-tolerance support, which is discussed in detail in this thesis. Centralization may also lead to a communication/performance bottleneck. In our case, this is not a problem
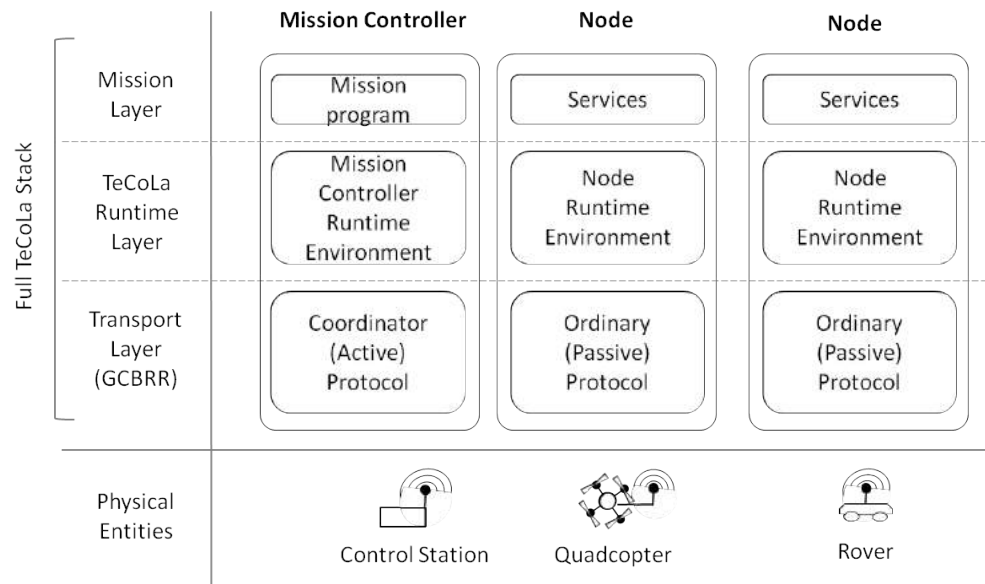
Figure 2.4: The TeCoLa software stack.

given that we target loosely-synchronised coordination scenarios where the communication between the mission controller and the nodes is not expected to be frequent or heavyweight.

2.3 Programming Support

The centralized coordination model is reflected in the software (middleware) stack we have developed to support application programming, called TeCoLa. The TeCoLa stack, illustrated in Figure 2.4, consists of three layers: the mission layer, the runtime system layer, and the transport layer.

Following the role separation between the mission controller (master) and the nodes (slaves), the mission layer for the former consists of the program that coordinates the mission, whereas for the latter it includes the services provided by the nodes. In the same vein, the mission controller features the runtime environment that is responsible for the execution of the mission program, while nodes feature a different runtime environment that interacts with the mission controller runtime to enable remote access of the node's services. At the transport layer, we employ the GCBRR protocol, which we have developed to support 1-N request/reply interactions in broadcast-based networks, in an efficient, coordinated way. The coordinator (active) part of the protocol, which initiates and orchestrates these interactions, runs on the mission controller, and the ordinary (passive) part of the protocol runs on the nodes.

In a typical deployment, the TeCoLa mission controller stack will be installed in a ground control station, while each of the UVs will feature the TeCoLa node stack. Note, however, that our design is completely modular, and allows the mission controller stack and node
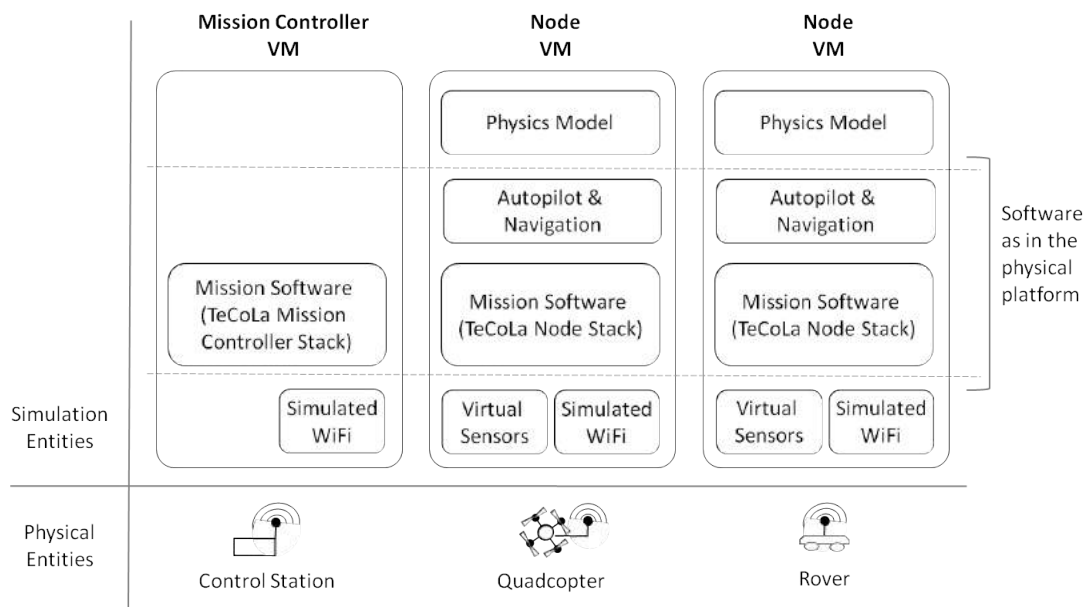
Figure 2.5: Approach for testing UV-based applications via simulation.

stack to reside on the same physical entity, thereby enabling hybrid deployments. For instance, the role of the mission controller could be assigned to a powerful UV, which has sufficient computing resources and is more robust/protected and thus unlikely to fail during the mission. As another option, the ground station could also act as a node, if it features certain sensors or actuators that are useful for the mission.

2.4 Testing Support

Testing is a key part of software development, even more so for software that controls/coordinates UVs, which has to be extremely mature before attempting any trials in the field. To enable a flexible and controlled testing of such software, we have designed and implemented a simulation environment where one may deploy and run the both system-level and application-level software in the same way as in a real-world deployment.

Figure 2.5 gives a high-level view of the simulation approach. Each UV is implemented using a separate virtual machine (VM), which hosts the very same software as the real platform, including the autopilot and navigation software as well as any additional software that is needed to support the mission —in our case, this is the TeCoLa node stack. The same holds for the control station, except that typically the respective VM will not feature any UV functionality and thus will only host the mission software —the TeCoLa mission controller stack.

The key difference between the real and the simulated platforms lies in the lower-level emulation of the sensors and actuators of the UVs. The respective VMs feature a physics

engine that reproduces the dynamics of the vehicle using a suitable model which captures the physical behavior of the vehicle when various forces are being applied to it. The physics engine, in turn, feeds this data to the autopilot, which processes the data and takes control decisions exactly as this is done in a real UV.

Besides the virtual sensors that are required by the autopilot, the VMs of UVs can be equipped with mission-specific virtual on-board sensors, such as cameras, which provide data to the mission software. Furthermore, the communication between the UV and control station VMs is supported over a simulated wireless network that is accessed by the mission software in a transparent way via the usual wireless interface. It is thus possible to test the same (unmodified) mission software (in our case, the full TeCoLa stack, including the mission program that coordinates the mission and the services that are provided by the nodes) using the same deployment configuration that is planned for the real-world deployment.

Chapter 3

Programming model

This chapter introduces the TeCoLa programming model. We start by presenting the main concepts that underpin the design of TeCoLa. Then, we describe a prototype implementation, including the concrete runtime objects and programming primitives/constructs introduced to support the programming of missions, and show how these can be used to write an indicative mission program. Finally, we discuss the overhead of the most critical programming primitives and the related work.

3.1 Basic Concepts

As discussed in the previous chapter, TeCoLa is designed having in mind a centralized coordination model where a distinguished entity, the mission controller, runs a program that monitors the state of the unmanned vehicles (UVs), analyses the situation, takes decisions about the actions that need to be performed to achieve the mission goals, and sends out corresponding commands to one or more UVs. The goal of TeCoLa is to simplify the development of such programs, also referred to as mission programs or applications.

Next, we discuss the main concepts behind TeCoLa. To separate the various concerns, for now, we do not get into the details of networking and transport protocols. We assume that the mission controller is able to discover and communicate with all UVs, using suitable communication support. In the same vein, we assume that if the mission controller fails then the mission stops and all UVs enter a fail-safe state. The provision of reliable group communication support over a wireless medium and the problem of tolerating failures of the mission controller are discussed in separate chapters.

Nodes and services

Each UV is abstracted as a *node* with a set of hardware and software resources. Hardware resources are the UV's CPU and other processing elements like DSPs, GPUs or FPGAs, as well as memories like RAM and Flash. In terms of software, apart from the operating system
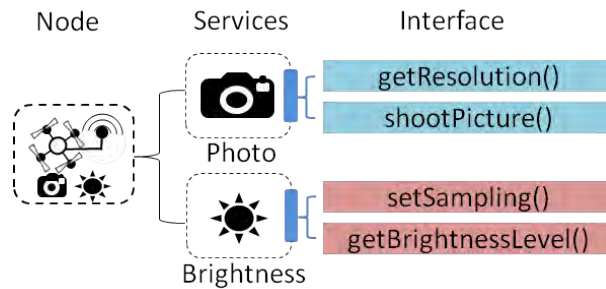
Figure 3.1: Mapping of resources to services.
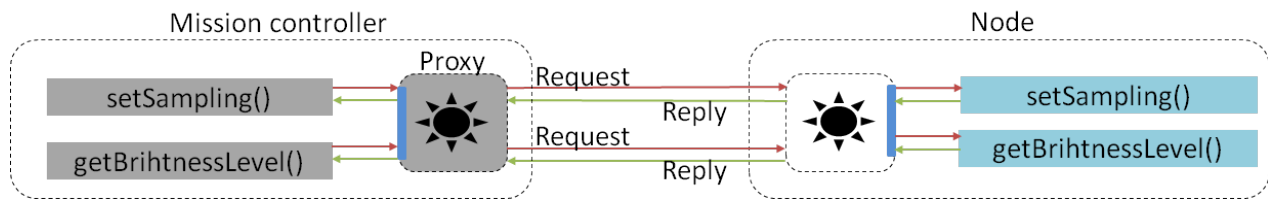
Figure 3.2: Information flow between a proxy and the node.

and drivers that are needed to talk to the hardware, the UV may include additional software modules/libraries that can be used to perform computations and data processing tasks.

Based on the available hardware and software resources, a node may provide one or more *services* to the outside world. In the spirit of service-oriented design, each service is accessed through a structured interface [51]. Without loss of generality, we let service interfaces consist of one or more *service calls*. Service calls can be used to issue commands to the nodes, or to retrieve state information from the nodes.

As an example, Figure 3.1 shows a node equipped with a camera and a brightness sensor, which map to two distinct services with indicative interfaces. The photo service interface includes the *getResolution*() call for retrieving the resolution of the camera, and the *shootPicture*() call for taking a snapshot. In the same spirit, the brightness service interface consists of the *setSampling*() call for setting the rate at which the sensor is sampled, and the *getBrightnessLevel*() call for retrieving the last value measured. Note that it is possible to have a service that relies on more than one resources. As an example, a camera could be combined with an image processing library to implement an object detection service that runs directly on the node.

Service access and failure semantics

The services of a node can be accessed remotely yet in a transparent way, through a *proxy* that features identical service interfaces (service stubs), as illustrated in Figure 3.2. The service calls made via a proxy work in the spirit of remote procedure calls (RPCs) and block until the node handles the request and returns the reply [33].

The nature of the operation that are requested via a service call can be either synchronous or asynchronous. Synchronous operations are performed immediately, and the result is returned in the reply of the respective service call. Asynchronous operations typically take a longer time to be performed. In this case, the reply of the call merely acknowledges the receipt of the request, and effectively serves as an indication 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. It is important to note that, unlike asynchronous procedure calls also referred to as promises [77], the remote service calls themselves are always synchronous and block the caller until the reply is returned, irrespectively of whether the operation they trigger in the node is synchronous or asynchronous.

Service calls have at-most-once semantics [76]. If a node fails before sending a reply, the mission controller (client) will receive an error, but it does not know whether the node actually executed the call. However, 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 [45].

Note that the mission controller may fail in the midst of a mission. From an RPC perspective, this corresponds to a client failure, which is typically handled by garbage collecting orphan calls at the server, using a suitable mechanism like extermination or reincarnation [96]. Contrary to this approach, in TeCoLa, service calls are not terminated abruptly. If a node detects a failure of the mission controller, it executes any pending calls to completion, and then enters a fail-safe state. This is done by notifying each local service so that it can take the necessary actions, including the handling of any ongoing asynchronous operations. Depending on the nature of the service, some asynchronous operations may be completed despite the failure of the mission controller, whereas others may be aborted when entering the fail-safe state.

Service heterogeneity and dynamics

Nodes may have diverse hardware resources. For instance, some nodes may be equipped with additional or more powerful processing elements and larger memories. Other nodes may feature certain sensors/actuators that are too expensive or too specialized for a certain function, and thus are not available in all nodes. Also, nodes can be heterogeneous in terms of software resources. As an example, some nodes may feature software for performing special data processing tasks, which, in turn, may rely on special hardware resources, such as FPGAs.

The aspect of hardware/software heterogeneity naturally extends to services. For instance, some nodes may provide the photo service (based on the camera resource), while other nodes may provide a spotlight service (based on a lamp resource). There can also be heterogeneity among services of the same class. As an example, some nodes may feature higher-resolution cameras or faster and more accurate image processing code than others. The different service classes and interfaces that are relevant for the application domain in

question can be defined in a formal/structured way using a suitable taxonomy/ontology [31]. Developers have to consult this ontology in order to write their missions so that they can exploit this service variety. The design of human- and machine-readable service ontologies is beyond the scope of this thesis. This problem has already been researched quite extensively in the context of web services, ubiquitous computing and the Internet of Things [22, 44, 55].

Note that the services provided by a node might vary in time. This can be due to dynamic resource availability or local resource management decisions. For instance, if the camera is damaged, the node will lose the ability to support the photo service. A node may also decide to power-down certain resources to save energy (e.g., the lamp), in which case it can no longer provide the services that depend on these resources (e.g., the spotlight service).

Node mobility as a service

Nodes represent UVs that can move in the field. We assume that nodes may rely on a common positioning system, such as GPS. We also assume that nodes come with basic autopilot capability, and can deal with low-level motor/control and obstacle avoidance issues in an autonomous way, in the spirit of [60]. Thus the mission program does not have to micro-manage nodes, and can control their movement in a high-level fashion, by specifying target locations or supplying intermediate waypoints, as usual in many robotic systems [2].

For reasons of uniformity, node mobility is also captured in a service-oriented way. The respective service interface may include service calls for setting a destination, several way-points or instructing the node to move towards a given direction. It can also include service calls for setting orientation and speed. In turn, the current position, speed and orientation of the node can be retrieved through corresponding service calls.

Of course, the actual movement capability of a node depends on whether the node was designed to move on the ground, on the water surface, underwater or fly in the air. This is yet another dimension of heterogeneity, which can be captured by introducing different classes of mobility services and suitable interfaces. As is the case for every other form of service heterogeneity, here too, the application must be designed to handle/exploit it in a meaningful way.

Mission group and teams

The nodes that participate in a mission form the so-called *mission group*. This group can grow dynamically to include additional nodes in the course of the mission. It can also shrink in case some nodes leave or crash/fail. The formation of the mission group is controlled by the mission program, which decides which types of nodes/services are needed at any point in time to perform the various tasks of the mission, and accordingly invites new nodes to join or allows participating nodes to leave. The mission program can browse the mission group in order to inspect the available proxies and the services that are supported by each one of them.
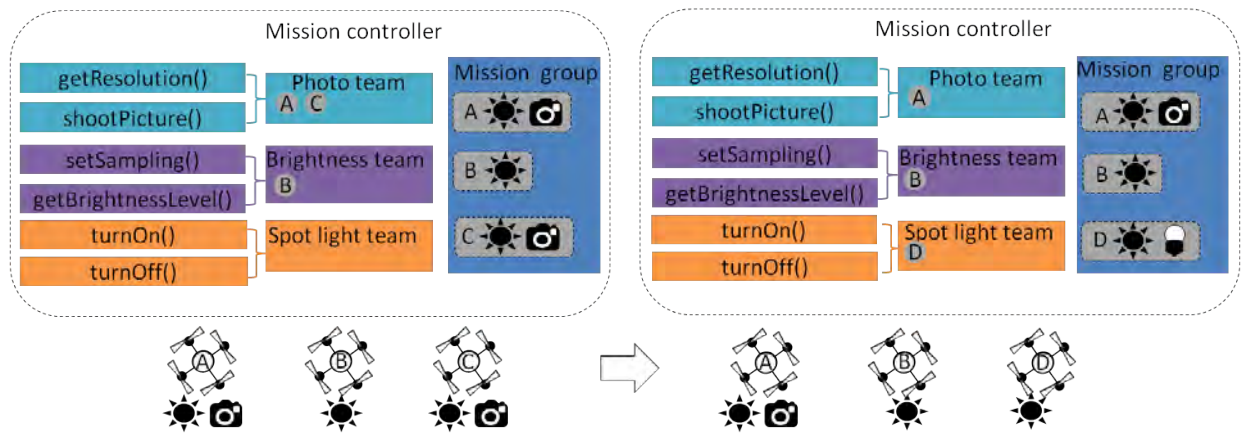
Figure 3.3: Team dynamics as a function of the current composition of the mission group.

A mission can be implemented by invoking the services of each node individually. However, this can become quite awkward when there are numerous nodes, especially if it is desired for several nodes to perform the same operation. For instance, one may wish to employ multiple UVs to scan a large area in order to detect specific situations or objects of interest.

To simplify programming, TeCoLa allows the formation and control of entire *teams*. A team can be created by adding specific, hand-picked nodes (manually-formed team), or by applying to the mission group a *membership* expression that refers to specific services (rule-based team) in which case all nodes that match the membership expression become members of the team (to avoid potential conflicts, a node cannot belong to different teams at the same time). It is possible to browse a team in order to inspect and invoke its members in the same way this is done for the mission group.

Manually-formed teams have to be maintained in an explicit way by the programmer. In contrast, rule-based teams are automatically maintained behind the scenes based on the nodes that are currently part of the mission group, without any intervention from the programmer. For instance, if a node joins the mission and its services satisfy the membership expression of a rule-based team, its proxy is automatically added to that team. Conversely, if a node no longer matches the membership expression of a rule-based team because it stopped providing a service, the proxy is removed from the team. Both manually-formed and rule-based teams are updated automatically if a member node fails, in whih case the node is removed from the team. Note that a team can be empty, if it ends up with no members, but the team still exists as an entity.

As an example, Figure 3.3 illustrates the evolution of three teams, for nodes with a photo service, a brightness service and a spotlight service, respectively. Initially, the mission group contains the nodes A and C which provide the photo and brightness service, and node B which only provides the brightness service. Hence the photo team has two members (A and C) and the brightness team has one member (B), whereas the spotlight team is empty.
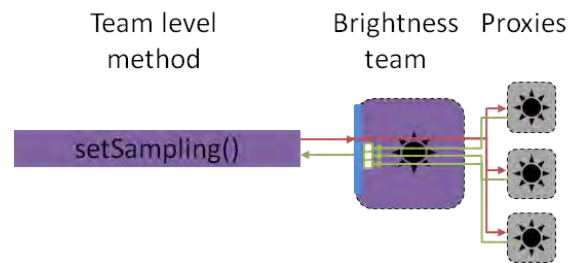
Figure 3.4: Team-level promotion of service calls.

Then, node C leaves the mission group and is removed from the photo team while node D with brightness and spotlight services joins the mission group, and as a result becomes a member of the spotlight team.

Services common to all members of a team are *promoted* to the team level. Service promotion has some analogy to inheritance via sub-classing in object-oriented models [58], but works in a bottom-up rather than top-down way. This is indicated in Figure 3.3 by depicting interface icons next to the team membership lists. Promotion leads to identical service calls at the team level, with the same name, arguments and return values as the service calls of the common service interface. The invocation of a team-level service call fans-out to the invocation of the respective service call in all proxies. This way, the mission controller can drive the entire team as if it were a single node. Notably, promoted service calls return a vector of replies, one for each team member. By iterating over this vector, one can retrieve the replies of individual team members. Figure 3.4 illustrates this principle for the brightness service team in Figure 3.3, assuming three proxies (team members).

3.2   Software organization

This section presents a prototype implementation of TeCoLa along the lines of the concepts described in the previous section. To accelerate development, instead of introducing a full-fledged language with its own runtime environment, we implement the functionality of TeCoLa on top of a standard Python environment (for a Linux platform) via specific classes that can be used by the application, without touching lower system-software layers or making changes to the Python runtime. We also keep the prototype as simple as possible in terms of special syntactic constructs in addition to what is already supported by the Python language.

Figure 3.5 shows the software organization of our prototype. Reflecting the master-slave approach, TeCoLa is internally structured in two runtime environments, the node runtime and the mission controller runtime, respectively. Note that the mission controller runtime can co-exist with the node runtime on the same physical entity/machine. The mission controller runtime is responsible for the execution of the mission program, performing under the hood all the necessary interactions with the nodes. The node runtime supports the interaction
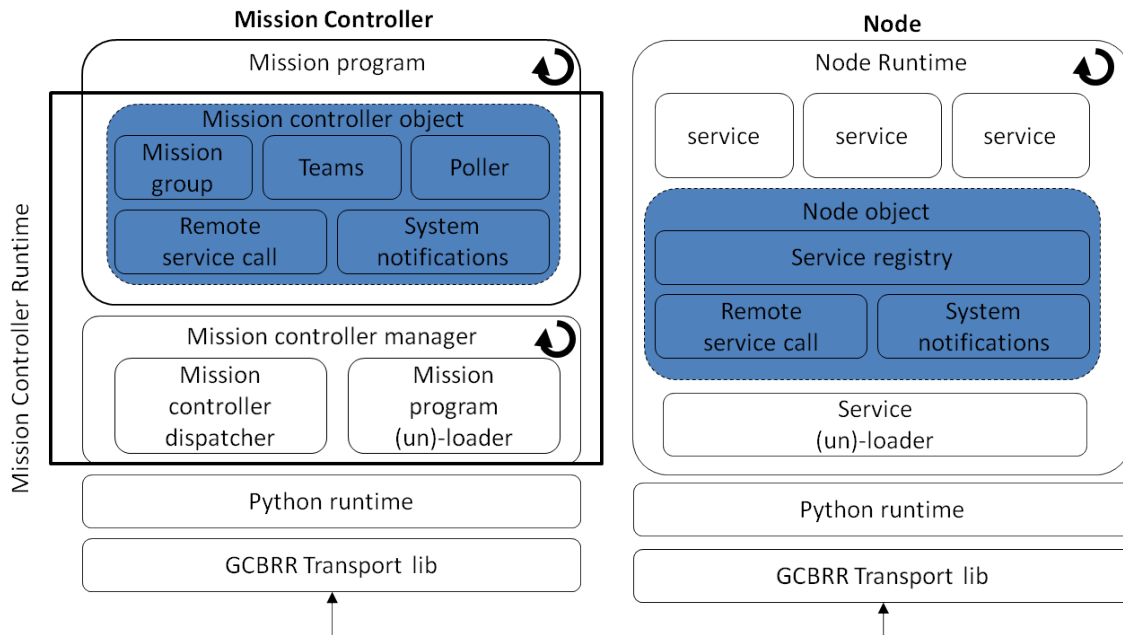
Figure 3.5: Software organization of the prototype implementation of TeCoLa.

with the mission controller, invoking the local services as requested. Figure 3.6 provides an overview of the node runtime operation, and the main functions that are activated as a result of its interaction with the mission controller runtime: (i) joining the mission group, (ii) service advertisement, (iii) service invocation, (iv) leaving the mission group, (v) handling and recovering from a failure of the mission controller. The latter is discussed in more detail in Chapter 4.

The communication between the mission controller runtime and the node runtime is performed using the GCBRR transport protocol, which provides support for coordinated group management and efficient 1-N request/reply interaction on top of a wireless medium. In particular, the mission controller runtime maps every remote operation including node- or team-level method invocation to a corresponding 1-N request/reply interaction, addressing the nodes that need to be involved in each case. In addition, special periodic request/reply interactions are performed by the mission controller runtime in the background, without any explicit request from the mission program, to confirm the aliveness of the nodes that are part of the mission group and to update the mission group in case of node failures. As part of this interaction, the mission controller retrieves updates about the services provided by each node. Last but not least, the mission controller uses the built-in support of GCBRR to let new nodes to join the mission group dynamically, as well as to remove nodes from the mission group in a controlled way. The details of GCBRR are discussed in Chapter 5.
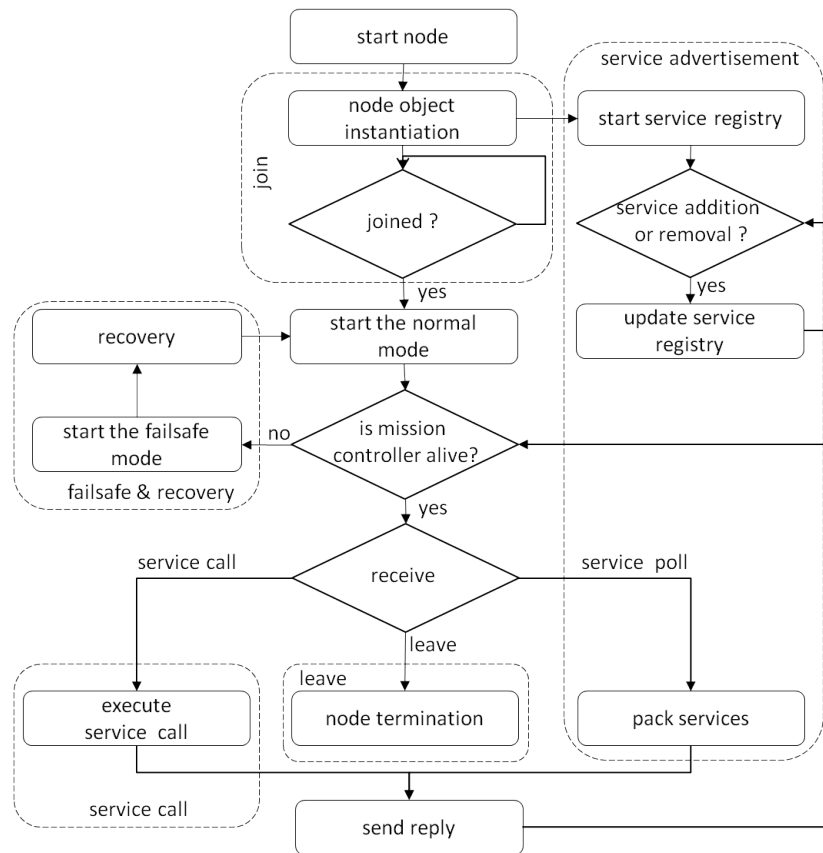
Figure 3.6: Operation of the node runtime.

Node runtime

The node runtime, shown in the right part of Figure 3.5, supports the operation of the node along the lines of Figure 3.6. It consists of a single process that employs a *Node* object which is an instance of the singleton *Node* class. This object contains most of the runtime functionality, and is responsible for advertising the services of the node, receiving the service call requests from the mission controller, executing them, and returning the replies. In particular, it implements three main mechanisms, the service registry, the remote service call, and the system notification mechanism.

The service registry maintains a description for each available service, that includes the service name, the signatures of the provided service calls and a reference to the service instance. The set of services that are initially loaded is specified in a configuration file. It is possible to add and remove services at any point in time. The contents of the registry are returned to the mission controller runtime as part of the response to a join and aliveness request.

The remote service call mechanism intercepts the calls of the mission controller that arrive via the transport layer, de-serializes each call request, and performs a lookup on the

service registry in order to find and invoke the respective service. At the end of each service call execution, it serializes the results and returns them to the transport for transmission to the mission controller. The node runtime executes one service call at a time while it is assumed that service calls execute fast, and do not block the node process. A service may still implement long-running operations in an asynchronous manner, using threads. In the current implementation, services run within the node process. As a consequence, if a service crashes, the entire node process will crash too. This is an implementation detail, and the issue can be addressed in a straightforward way by running each service within a separate process or using container-technologies.

The *Node* object is also responsible for handling system notifications that are produced internally by the node runtime. More specifically, the system notification mechanism of the *Node* object intercepts and handles the notifications produced by the mechanism that loads and respectively unloads individual services, and invokes the corresponding service activation and deactivation routines. It also intercepts the notification that is produced by the transport layer when a failure of the mission controller is detected, and invokes the fail-safe operations of the services that are currently loaded/active. Finally, the administrator of the node runtime can inspect the loaded services through the same mechanism.

Finally, the node runtime performs the necessary operations for the node to join the mission group, in a transparent way. More specifically, after the instantiation of the *Node* object, it waits for an invitation of the mission controller in order to join the mission group so that it can starting receiving/executing service calls. In a similar vein, when the mission controller asks the node to leave the mission group, the node is brought to a fail-safe state in a similar way this is done when detecting a failure of the mission controller.

Mission Controller runtime

The mission controller runtime is responsible for running the mission program. As shown in the left part of the Figure 3.5, the functionality of the mission controller runtime is split in two processes, the mission program process (program process) and the mission controller manager process (manager process). This is done to have a clean separation between the mechanisms that are used for the execution of the mission program and the mechanisms that are used to initialize and manage this execution as well as to recover from a failure of the mission controller.

The main responsibilities of the manager process include the loading, execution and termination of the program process. The manager process also acts as a mediator between the program process and the transport layer. During the loading of the mission program, a functionality which is performed by the mission program (un)-loader mechanism, the manager process spawns the program process and informs the dispatcher to start forwarding messages between the mission program and the transport. At the termination of the mission program, the manager process suspends the dispatcher and terminates the program process. The dispatcher forwards the remote service requests of the mission program to the transport layer, and returns the node replies that arrive from the transport back to the mission program. The

dispatcher also intercepts and forwards to the mission program system notifications regarding the aliveness of the nodes as well as the termination of the mission program. Similarly to the node runtime, the administrator of the mission controller can inspect the status of the mission program through the dispatcher mechanism.

The mission program invokes the functions of TeCoLa via the *MissionController* object, which is an instance of the singleton *MissionController* class. This object is instantiated from the mission program and lives in its address space. The *MissionController* object interacts with the transport through the dispatcher via a standard unix socket. It implements the remote service call mechanism which maps every node- or team-level service call to a corresponding request/reply interaction, addressing the nodes that need to be involved in each case and serializes the requests and de-serializes the respective replies. In addition, it creates proxy objects for the nodes that join the mission, and populates the mission group as well as the team objects accordingly. It also implements the logic behind all the waiting operations. This is done via polling, using a mechanism that issues periodically the service calls of the waiting conditions in the background without any explicit request from the mission program.

3.3 Programming primitives

This section shows how the different concepts map to programming primitives, and illustrate their usage via short code excerpts. We adopt an object oriented flavor, and we take advantage of various features of Python such as the dynamic types in order to create suitable syntactic definitions and constructs. Python was chosen because of its popularity among developers, however the TeCoLa primitives could be supported on top of any other programming language with similar features.

Services

Services are implemented as independent software components (singleton classes), derived from the *Service* meta-class. Each service exposes its functionality through a number of service calls, which can be invoked remotely via the respective proxy. As already discussed, the service class names, calls, arguments, return values and semantics, would have to be formally defined via a suitable taxonomy/ontology.

Listing 3.1: Skeleton of a service implementation.

```python
class BrightnessSvc(object):
    __metaclass__ = Service

    def getBrigthnessLevel():
        return brigthnessLevel

    def setSampling(rate):
```

```
 8        set_sampling_rate(rate)
 9
10    def __activate():
11        turn_on_brightness_sensor()
12        set_sampling_rate(defaultrate)
13
14    def __passivate():
15        turn_off_brightness_sensor()
16
17    def __failsafe():
18        turn_off_brightness_sensor()
```

To give an example, Listing 3.1 provides a skeleton for a brightness service that offers a simple interface in the spirit of Figure 3.1 (the *object* class parameter is Python specific). Note that every service has to provide three mandatory methods: the *activate*() and *passivate*() methods for activating and respectively passivating the service, as well as the *failsafe*() method that is invoked to bring the service in a fail-safe state.

Note that a service may internally use threads to implement background activities and asynchronous operations. For instance, the brightness service could use a thread to periodically read the brightness sensor according to the specified sampling rate, and update the brightness property accordingly. The details depend on the hardware components, drivers and user-level libraries used to implement the respective functionality. In any case, we view services as distinct components that can be developed from scratch, downloaded/re-used from open-source repositories, or come pre-installed as part of the stock software/firmware that ships with a given node (UV platform).

Mission Controller object

This object supports different methods that can be used by the mission program to populate and to inspect the mission group, form teams and control individual nodes but also entire teams, as needed. These aspects are discussed in more detail below.

Mission group

The mission group is maintained as part of the *MissionController* object, and is accessed through the *group* collection. It contains the proxies for all nodes that currently participate in the mission group.

The mission group is populated when the mission programmer decides to allow new nodes to join the mission group. This is done through the *checkJoin*() method of the *MissionController* object, which is basically a wrapper for the corresponding operation of the GCBRR transport, which broadcasts a join request and waits to receive replies from the nodes that wish to join the mission group. The mission programmer supplies the waiting time of the join operation as an argument. The *checkJoin*() method could be easily extended

to support more targeted node discovery based on services, node names or types, however this is left as a future work.

In the same vein, the mission programmer may enforce the removal of operational nodes from the mission group, through the *forceLeave*() method of the *MissionController* object. This method is also a wrapper for the corresponding operation of the GCBRR transport. It takes as an argument an array that contains the names of the nodes to be removed, and leads to their removal.

If a node crashes/fails, it is removed from mission group automatically, without this being requested in an explicit way by the mission program. More specifically, the mission controller confirms the alivness of the nodes of the mission group through periodic null requests that serve as heartbeats in the absence of requests from mission program. The period as well as the number of the required missing heartbeats in order for a node to be declared as failed are passed as parameters to the MissionController object. Note that such configuration is transfered to each node when it joins the mission group in order to allow nodes to detect failures of the mission controller.

Listing 3.2 shows an example where the mission controller runtime is configured through the *MissionController* object, to check for the aliveness of the nodes of the mission group every 2 seconds and to declare a node failure after 3 missing heartbeats. Note that the aliveness check configuration can also be changed at runtime, via the *setAlivnessChk*() method, which takes the same parameters as the constructor of the *MissionController* object. Next, the mission program performs the *checkJoin*() operation and waits for 2 seconds for new nodes to join the mission group. Later on, the mission program invokes the *forceLeave*() method to force nodes A and B to leave the mission group.

Listing 3.2: Invite nodes to join the mission group and to leave .

```
1
2  mc = MissionController(2, 3)
3
4  mc.checkJoin(2)
5
6  ...
7  try:
8    mc.forceLeave(["A","B"])
9  except NodeNExist:
10   mc.exit()
```

If the nodes that are instructed to leave the group are not part of the mission group, the *forceLeave*() method raises a runtime exception. In the given example, the mission program catches this exception and terminates.

Handling group updates

The mission program may wish to be notified about group changes that take place during the mission, when nodes join, leave or fail. To this end, it can install a handler which is invoked automatically each time the mission group changes. Group update handlers take as parameters four collections/lists, containing respectively the current mission group, the nodes that joined the group, the nodes that left the group, and the nodes that have failed. For the left and failed nodes, the collections contains reduced proxy objects, with only the name and type of the node but without any service inerfaces (it is not possible to remotely invoke nodes that no longer belong to the mission group). These reduced proxies are garbage collected right after the completion of the handler.

Listing 3.3: Writing and installing group update notifiers.

```python
def GroupUpdate(grp, joined, left, failed):
    print "Count:" + str(grp.size())
    for node in joined:
        print "joined " + node.name
                        + ":" + node.type

    for node in left:
        print "left " + node.name
                        + ":" + node.type

    for node in failed:
        print "failed " + node.name
                        + ":" + node.type

mc = MissionController(...)
mc.group.setUpdateHandler(GroupUpdate)
```

Listing 3.3 gives an example where the mission program uses a group update handler to print the type and name of nodes that joined/left the group or have failed.

Inspection of the mission group and node invocation

The mission program can traverse the *group* collection using standard Python iteration structures, in order to see which nodes are part of the mission group as well as to inspect the services provided by each node of them. Individual nodes can then be controlled by invoking their services through the corresponding proxy objects.

Listing 3.4: Browsing the mission group and invoking node services.

```python
mc = MissionController(...)

for n in mc.group:
```

```
4      for svc in n.services:
5        if isinstance(svc,PhotoSvc):
6          picture = svc.shootPicture()
7
8    for n in mc.group:
9      try:
10       n.PhotoSvc.shootPicture()
11     except SvcError:
12       mc.exit()
```

An example is given in Listing 3.4, where the mission program takes pictures via the nodes that provide the photo service. The last piece of code shows how to do the same by invoking the service without checking that it is actually supported. If the node does not support the service, a runtime exception will be raised, which is caught by the mission program (in this case, the program decides to terminate).

Waiting on asynchronous operations

As mentioned, a service may implement asynchronous operations, which may take a long time to complete. Besides offering a call for starting such an operation, to be able to check its progress, the service also has to provide a separate call that returns its status information.

It is often desirable for the mission program to block until a certain state is reached. This can be done via the *wait*() method of the *MissionController* object, which takes as a first parameter an array of conditions. Each condition is a given as a tuple that consists of a service call, a comparison operator, and a value. The *wait*() operation blocks until all conditions becomes true[1]. Behind the scenes, the nodes are polled via periodic invocations of the respective service calls, and the return values are compared with the respective value of the condition. The programmer supplies the desired polling period in the second argument of the *wait*() operation. The third argument is a timeout, after which the *wait*() operation returns with an exception.

Listing 3.5 shows code that uses the mobility service in order to instruct a node to move to a certain position, waits until the node reaches that position, and then takes a picture via the photo service. In this case, the condition expression refers to the *getDistanceFromTarget*() call of the mobility service, which returns the geometrical distance from the target position that was specified via the last *goto*() call. Also, it suffices for the node to reach the target position with a tolerance of 1.0 meters. The invocation of the *getDistanceFromTarget*() service call and the evaluation of the condition expression is performed at the specified period of 10 seconds, while the mission program is prepared to wait for at most 100 seconds. If the condition is not satisfied within that interval, the *Timeout* exception will be raised

---

[1]The wait statement could be more elaborate in order to support more complex conditional expressions. Since this is not very interesting for our research, we keep the prototype implementation as simple as possible in this respect.

and the node will be instructed to land. If the node fails during the invocation of the *getDistanceFromTarget*() service call, the *NodeFailure* exception will be raised and the mission program will terminate.

Listing 3.5: Waiting on conditions.

```
1  # n is a node proxy
2
3  n.MobilitySvc.goto(pos)
4
5  cond = [[n.MobilitySvc.getDistanceFromTarget ,< ,1.0]]
6  try:
7     mc.wait(cond,10,100)
8  except Timeout:
9     n.land()
10 except NodeFailure:
11    mc.exit()
12
13 print n.MobilitySvc.getCurPos()
14 pic = n.PhotoSvc.shootPicture()
```

As a special case, the programmer may specify a polling period that is equal to zero. Then, polling will be performed based on a default polling period, which the mission program can set/change at any point in time through a corresponding method of the mission controller object. The mission program may also supply a negative timeout value, in which case the *wait*() operation will block arbitrarily long until the condition is satisfied; if the condition is never satisfied, the mission program will deadlock at this point.

To have more flexibility, the mission program can block on several wait conditions until at least one of them becomes true. This is achieved via the *select*() method of the *MissionController* object, which provides similar functionality to that of the select statement found in other languages and programming environments, such as CSP [62] and Plan9 [85]. More specifically, the *select*() method takes as a first parameter a list containing the different cases to be considered (if this list is empty, an exception is raised). Each case is a tuple that consists of a label and an ordinary wait condition. When the *select*() operation is invoked, the runtime evaluates automatically the wait conditions one by one. As soon as a condition becomes true, the *select*() operation unblocks, the label of the respective case is returned to the mission program, and the case is removed from the list. Similar to the *wait*() method, the programmer supplies the desired polling period and the timeout, in the second and third argument, respectively.

Listing 3.6: Waiting on multiple conditions.

```
1  # n1 and n2 are node proxies
2
3  n1.MobilitySvc.goto(posA)
```

```
4    n2.MobilitySvc.goto(posB)
5
6    condA = [[n1.MobilitySvc.getDistanceFromTarget,<,1.0]]
7    condB = [[n2.MobilitySvc.getDistanceFromTarget,<,1.0]]
8
9    select_cond=[["N1_WP",condA],["N2_WP",condB]]
10
11   While True:
12     try:
13       label = mc.select(select_cond,10,100)
14
15       if label == "N1_WP":
16         print n1.MobilitySvc.getCurPos()
17         picA = n1.PhotoSvc.shootPicture()
18       elif label == "N2_WP":
19         print n2.MobilitySvc.getCurPos()
20         picB = n2.PhotoSvc.shootPicture()
21
22     except Timeout:
23       l = mc.getCurSelectLabel()
24       if l == "N1_WP":
25         n1.land()
26       elif l == "N2_WP":
27         n2.land()
28     except NodeFailure:
29       l = mc.getCurSelectLabel()
30       print "Select Condition %s" % (l)
31       mc.exit()
32     except EmptyCndList:
33       n1.land()
34       n2.land()
35       mc.exit()
```

Listing 3.6 extends the example of Listing 3.5, and shows code that instructs two nodes to move to two different positions and to take a picture when the desired positions are reached. In this case, it is not desirable to wait for the first node to reach the desired position in order to take a picture, and then to do the same for the second node, in a sequential fashion. It also not desirable to wait for both nodes to reach their positions, before any of them takes a picture. Instead, it is desirable to take pictures as soon as possible. For this purpose, the mission program uses the *select*() method to invoke the photo shooting operation of whichever node happens to reach its position first. When this is done for both nodes, the case list becomes empty and the corresponding exception is caught in order to instruct the nodes to land before terminating the mission program.

Team formation and team invocation

Different teams can be introduced via the $createTeam()$ method of the $MissionController$ object, which returns a $Team$ object. The method takes as input a so-called membership expression, which consists of three parameters, with each one serving as a selection filter that is progessively applied in a pipelined-fashion, starting with the mission group. The first parameter selects from the mission group the nodes with certain names, the second parameter refines the selection in order to pick nodes of a certain type, and finally the third parameter selects the nodes that feature certain services. Each parameter represents a logical expression, with the individual terms being stored in a list. For the first and the second parameter, the terms are interpreted as parts of a logical OR expression of the form $term_1 \vee term_2 \vee ... \vee term_n$. The terms of the third parameter are interpreted as parts of a logical AND expression of the form $term_1 \wedge term_2 \wedge ... \wedge term_n$, in order to guarantee that all team members will feature the services that are used as a selection criterion, and thus these will also be promoted to the team level.

When a node updates the services it provides or when it joins or leaves the mission group, all membership expressions are automatically re-evaluated and the corresponding teams are updated accordingly to contain the right proxies. It is possible for a node to match several team membership expressions, however it will be added only to the first team that matches (the membership expressions are evaluated internally one by one). Furthermore, the programmer can add nodes (proxies) to a team in a manual way, via the $addNode()$ method of the team object (if the node is already a member of a team, an exception will be raised). It is important to note that a node which is added manually does not have to match the team membership expression; no compatibility check is performed in this case, and it is the programmer's responsibility for such additions to makes sense. Also, manually added nodes are persistent, in the sense that they are not removed from the team unless the programmer does this explicitly, by invoking the $removeNode()$ of the team object, or the node fails, in which case the proxy is removed from the team automatically.

To have more flexibility, the programmer may use the special keyword *any* as a selection criterion to match/select all available nodes. For instance, if all parameters of the $createTeam()$ method contain the keyword *any*, then the returned team object will contain all the proxies of the mission group. On the contrary, using an empty selection expression in any of the selection parameters forces an empty selection, in which case the $createTeam()$ method will return an empty team. Note that this makes it possible to create teams that are managed purely in a manual way; initially, an empty team is created by using at least one empty selection criterion, and then it is populated by manually adding and/or removing specific nodes, as discussed above.

Teams can be destroyed with the $destrTeam()$ method of the $MissionController$ object. This does not involve any communication with the nodes, and always succeeds even if the team is currently empty. Note that the invocation of the $destrTeam()$ method does not affect the long-running operations that have been issued in the context of a team (see next). Provided that there is a suitable service call for this, such operations can be explicitly

stopped by the mission program, if desired, either by invoking that call at the team level before destroying the team, or by invoking it for the individual nodes that were part of the team when the operation was started.

Listing 3.7 shows how a team with one member can be formed manually based on the node's name. In this example if the node has a membership conflict the team is destroyed.

Listing 3.7: Manual team creation.

```
1  team = mc.createTeam(["none"],["none"],["none"])
2
3  try:
4     team.addNode("NodeA")
5  except NodeNExist:
6     print "Node is not member of the mission group"
7  except MembershipConflict:
8     print "Node belongs to another team"
9     mc.destrTeam(team)
```

The mission program can browse the team and install an update handler for the corresponding membership updates, as this is done for the mission group. However, the team update handler is not invoked for node additions/removals that are performed explicitly by the programmer via *addNode*() and *removeNode*(). Listing 3.8 shows how to create a rule-based team for nodes that provide the photo service. Also, a team update handler is installed, which prints the type and name of the team members.

Listing 3.8: Creation of a rule-based team with an update notifier.

```
1  def PrintTeam(team,added,removed):
2     print "Count:" + str(team.size())
3     for n in team
4        print n.name + ":" + n.type
5
6  team = mc.createTeam(["any"],["any"],["PhotSvc"])
7  team.setUpdateHandler(PrintTeam)
```

Services that are common among the members of the team are promoted to the team level, and the respective interfaces become directly accessible via the respective *Team* object. Note that it can be safely assumed that a team will *definitely* feature the services used as a selection criterion in the respective membership expression. Therefore a team-level service call will always succeed (except for the special case where the team happens to be empty, in which case an exception is raised. Service promotion also applies to other services that might be provided by all team members but are not in the membership expression. However, such *opportunistically* promoted services cannot be assumed as given, and have to be discovered explicitly in order to be invoked.

Team-level services can be inspected in the same way this is done for individual nodes (proxies). The invocation of a team-level service call transparently leads to the invocation of the respective service calls at all members of the team. The results are returned as a dictionary of key-value pairs, where the key is the node proxy and the value is the result of the respective service call. Notably, service promotion allows the programmer to control an entire team in a straightforward way, almost as if it were a single node. Note that during the execution of a team update notifier all team-level methods of this particular team outside the notifier will block at the invocation until the completion of the notifier.

Listing 3.9: Team-level control.

```
1  # team has already been created
2
3  team.MobilitySvc.relativegoto(pos);
4  cond = [[team.MobilitySvc.getRelDistanceFromTarget,<,1.0]]
5  mc.wait(cond,10,100)
6
7  print team.MobilitySvc.getCurPos()
8  pictures = team.PhotoSvc.shootPicture()
9
10 for node,pic in pictures.iteritems():
11    detect_something(pic)
```

Listing 3.9 shows how to move the photo team (created in Listing 3.8) at a certain location, print the positions of individual members, and let all team members take a picture. Note the similarity with Listing 3.5 which does the same for a single node. In this case however, instead of having all team members go to the same position via the *goto*() call, the movement of the team is done via the team-level *relativegoto*() call. The difference is that each team member interprets the specified position relatively to an offset, which can be set by the mission program to a different value for each team member (see next). In the same vein, instead of using *getDistanceFromTarget*(), the distance of each team member from the target team position is retrieved via the *getRelDistanceFromTrarget*() call.

The mission program can set the position offset via the *setPosOffset*() call of the mobility service at any point in time. This is used internally by the mobility service to determine the target position of the node when instructed to move via the *relativegoto*() call, as well as to calculate the distance from the target position that is returned via the *getRelDistanceFromTrarget*() call. The position offset can be set to a fixed value, or it can be adjusted in a dynamic way in order to maintain a certain formation based on the number of nodes that are part of the team, by programming the team update notifier accordingly.

Listing 3.10: Formation maintenance pattern in a dynamically changing team.

```
1  def UpdateFormation(team,refpos):
2     poslist = calcFormationPositions(refpos,formation_pars,team.size())
3
```

```
4      pos_iter = itertools.cycle(poslist)
5      for n in team
6        pos = next(pos_iter)
7        offset= calcOffset(refpos,pos);
8        n.MobilitySvc.setPosOffset(offset)
9
10     team.MobilitySvc.relativegoto(refpos);
11     cnd = [[team.MobilitySvc.getRelDistanceFromTarget,<,1.0]]
12     mc.wait(wp_cnd,10,100)
```

Listing 3.10 shows a generic pattern that can be used to maintain/adapt a team formation with reference to a single team-level position that is supplied as a parameter. Such code can be included as part of the team update handler. As a first step, the individual positions for the team members are calculated as a function of the reference position, the formation parameters, and the number of team members, via the *calcFormationPositions* helper routine. Then, the position offset relative to the reference position is calculated and re-set for each team member. Finally, the entire team is instructed to move to the reference position, and as a result each member will move to its own position in the context of the formation. The update of the formation completes when all team members reach their positions.

3.4 Application Example

In the following, we show how TeCoLa is used to implement a forest fire detection and response scenario, in the spirit of the application example that is discussed in Section 2.1. The application uses two different types of drones: the scanner and the fire extinguisher. Each type comes with different flight capabilities and equipment, and provides different special services. The scanner can fly long distances at high altitude, carries high-resolution infrared and visible light cameras, and features a service that processes the images of these cameras in order to detect the presence of a fire. This service reports the locations of the spots along with their intensity. The fire extinguisher is a drone equipped with a water tank, and provides a fire extinguisher service with a call for spraying/emptying the contents of the water tank, as well as a call for reloading the tank (e.g., by pumping water from a lake/pool or the sea).

The target forest area is continuously monitored by scanners. At a certain location in the forest, there is a pavilion of the fire department where a number of extinguishers are stationed. When a fire outbreak is detected by the scanners, the fire extinguishers are sent out to drop water at the spots where the fire is most intense. After the drop, the fire extinguishers go to a specified location in order to reload their water tanks, and return to the fire spots. The scanners keep scanning the entire forest area as before, in order to update the spots that need to be handled with priority as well as to detect new fire spots (consecutive fire outbreaks at widely different locations are common in case of arson attempts). When

there are no more fire spots left, the fire extinguishers are instructed to go back to their station and land.

The mission program employs two teams, for the fire detection and fire extinguisher service, respectively. Each team maintains a different formation suitable for its purpose. More specifically, the scanner team performs a high-altitude flight, scanning the area of interest in consecutive passes, from one border to the other, until the entire area is covered. The number of passes that needs to be performed to cover the area depend on the size of the area, the number of drones in the scanner team and their flight altitude. The team of the fire extinguishers maintains a parallel line formation, and moves back and forth between the locations where the fire is most intense and the re-load area. To better illustrate how TeCoLa is used to implement this application, we selectively discuss the most important parts of the mission logic.

The main body of the mission program is shown in Listing 3.11. The program starts by instantiating the *MissionControler* object. The period for checking the aliveness of the nodes in the mission group is set to 2 seconds, and the number of consecutively missing heatbeats for declaring a node as failed is set to 3. Then, a team of scanners is created, using as selection criterion the provision of the *FireDetector* service , and the update notifier for that team is installed. Finally, the program enters the main loop, where it iteratively invokes the *select* operation and each time takes an action according to the returned label.

Listing 3.11: Forest fire detection and response mission program.

```
1  # association between labels and actions
2  ops={ "SCN_IN_POS": scannersInPosition ,
3      "SCN_CHK_FIRE": scannersCheckForFire ,
4      "FEXT_SPRAY_POS": fireExtInSprayPosition ,
5      "FEXT_LOAD_POS"  : fireExtInLoadPosition
6    }
7
8  # globals
9  select_cnds = []
10 scn_plan = None
11 scanners = None
12 ext_plan = None
13 extinguishers = None
14
15 mc = MissionController(2,3)
16
17 # scanner team
18 scannners = mc.createTeam([any],[any],[FireDector])
19 scannners.setUpdateHandler(scannerTeamUpdate)
20
21 # main loop
22 while True:
```

```
23    try:
24        label=mc.select(select_cnds,5,200)
25        ops[label]()
26    except EmptyCndList:
27        mc.checkJoin(5)
```

The association between labels and actions is defined via the *ops* dictionary. As will be discussed below, the cases of the *select* operation are updated in a dynamic way, by the actions that are executed each time and the update notifiers of the different teams. If the *select* is left without any cases, the mission program waits 5 seconds for new nodes to join the mission group. Note that this is what actually happens in the first iterations, until some scanners nodes join (the case list of the *select* operation is initially empty).

Listing 3.12 shows the update handler for the scanner team. Its purpose is to configure (or re-configure) the available scanners so that the target area is scanned by all of them in parallel to have a faster coverage of the target area. When the team changes, the scan plan is updated as a function of the target area, the flying altitude and the size of the scanner team, via the helper routine *updateScanPlan*(). This routine (not shown here for brevity) recalculates the "slices" of the target area that can be scanned by the team in a single pass and the seqeuence in which these slices should be scanned, and returns the updated plan. Then, the position offset for each team member is calculated, using as a reference the next waypoint of the scan plan, via the helper routine *calcScanPositions*. Finally, the team is instructed to move to that waypoint, and a corresponding wait condition is appended to the cases of the *select* operation that is invoked from the main loop.

Listing 3.12: Update handler for the scanner team.

```
1   def scannersTeamUpdate(team,added,removed):
2     # update plan and get next waypoint
3     scn_plan = updateScanPlan(scn_plan,config.area,config.altitude,team.
        size())
4     scanpos = next(scn_plan.wps)
5
6     # calculate and set offset for each member
7     poslist = calcScanPositions(scanpos,config.altitude,team.size())
8     pos_iter = itertools.cycle(poslist)
9     for n in team
10        pos = next(pos_iter)
11        offset= calcOffset(scanpos,pos);
12        n.MobilitySvc.setPosOffset(offset)
13
14    # move to next scan waypoint
15    team.MobilitySvc.relativegoto(scanpos);
16    cond = [[team.MobilitySvc.getRelDistanceFromTarget,<,1.0]]
17    select_cnds.append(["SCN_IN_POS",cond])
```

When the scanner team reaches the specified position, the select operation returns the *SCN_IN_POS* label, which leads to the execution of the *scannersInPosition*() routine (see label/action association in *ops*). The *scannersInPosition*() routine, shown in Listing 3.13, instructs the scanner team to move to the next waypoint of the scan path. It also adds a new case to the *select* of the main loop, in order to return the *SCN_CHK_FIRE* label when the team covers a certain distance (which is calculated/updated as part of the scan plan).

Listing 3.13: Scanners in position operation.

```
1  def scannersInPos():
2    # move to next scan waypoint
3    scanners.MobilityScv.relativegoto(next(scn_plan.wps))
4    cond = [[scanners.Mobility.getRelDistanceFromTarget,"<",1.0]]
5    select_cnds.append(["SCN_IN_POS",cond])
6
7    # return a check−fire label when a certain distance is covered
8    scanners.MobilityScv.resetDistanceMeter()
9    cond = [[scanners.Mobility.getDistanceMeter,">",scn_plan.dist]]
10   select_cnds.append(["SCN_CHK_FIRE",cond])
```

When the *SCN_CHK_FIRE* label is returned from the select operation, the mission program will execute the *scannersCheckForFire*() routine, shown in Listing 3.14. This invokes the *detect*() service call of the *FireDetector* team-level service, and adds a case for the *select* operation in order to trigger the next fire check. If the current check detects a fire outbreak, the mission program updates the plan that should be followed by the fire extinguisher team. If the team does not exist yet, it is created and assigned the respective team update handler.

Listing 3.14: Fire detection operation.

```
1  def checkForFire():
2    # get detection results
3    detected_spots = scanners.FireDetector.detect()
4
5    # return a check−fire label when a certain distance is covered
6    scanners.MobilityScv.resetDistanceMeter()
7    cond = [[scanners.Mobility.getDistanceMeter,">",scn_plan.dist]]
8    select_cnds.append(["SCN_CHK_FIRE",cond])
9
10   if detected_spots is not None
11     # update the plan for the fire extinguisher team
12     ext_plan = updateFireExtinguishPlan(ext_plan,detected_spots)
13     if extinguishers is None
14       # create fire extinguisher team and allow extra nodes to join
15       extinguishers = mc.createTeam([any],[any],[SpraySvc])
16       extinguishes.setUpdateHandler(fireExtTeamUpdate)
```

```
17        mc.checkJoins(5)
```

The rest of the mission logic is implemented in the update handler of the fire extinguisher team and in the routines that are invoked when the extinguishers reach the positions to empty and re-load their water tanks. Listing 3.15 shows the update handler of the fire extinguisher team, which is very similar to that of the scanner team.

Listing 3.15: Update handler for the fire extinguisher team.

```
1  def fireExtTeamUpdate(team, added, removed):
2    # get next spray waypoint
3    spraypos = next(exp_plan.wps)
4
5    # calculate and set offset for each member
6    poslist = calcLinePositions(spraypos, config.density, team.size())
7    pos_iter = itertools.cycle(poslist)
8    for n in team
9      pos = next(pos_iter)
10     offset= calcOffset(spraypso, pos);
11     n.MobilitySvc.setPosOffset(offset)
12
13   # move to next spray waypoint
14   team.MobilitySvc.relativegoto(spraypos);
15   cond = [[team.MobilitySvc.getRelDistanceFromTarget,<,1.0]]
16   select_cnds.append(["FEXT_SPRAY_POS",cond])
```

When the fire extinguisher team reaches the specified position the $fireExtInSprayPosition()$ routine which is shown in Listing 3.16 is invoked. The entire team performs the spray operation and then it is instructed to move to the load position to perform a reload.

Listing 3.16: Fire extinguishers in spray position operation.

```
1  def fireExtInSprayPosition():
2    # spray fire spot
3    extinguishers.Spray.spray()
4
5    extinguishers.MobilitySvc.relativegoto(ext_plan.loadpos);
6    cond = [[team.MobilitySvc.getRelDistanceFromTarget,<,1.0]]
7    select_cnds.append(["FEXT_LOAD_POS",ext_wp_cnd])
```

In the same spirit, when the fire extinguisher team reaches the load position, *select* will return the $FEXT\_LOAD\_POS$ label and it turn the mission program will execute the $fireExtInLoadPosition()$ routine, shown in Listing 3.17. This routine performs the team-level load operation, and instruct the team to move to the next spot. If the fire extinguisher plan contains no other fire spots, the team is instructed to return to base/land and is then destroyed.

Listing 3.17: Fire extinguishers in load position operation.

```python
1  def fireExtInLoadPosition():
2    # load
3    extinguishers.Spray.load()
4
5    if ext_plan.spray_wps:
6      # move to next spray waypoint
7      extinguishers.MobilityScv.relativegoto(next(ext_plan.wps))
8      cond = [[extinguishers.Mobility.getRelDistanceFromTarget,"<",1.0]]
9      select_cnds.append(["FEXT_LOAD_POS",cond])
10   else:
11     # teams returns to base and lands
12     extinguishers.MobilitySrv.rtl()
13     # destroy team
14     mc.destr(extinguishers)
```

This non-trivial functionality can be implemented in a clean and straightforward way using the TeCoLa primitives: teams, team update notifiers, team-level services, and the wait and select operations. The application logic is about 100 lines of code without the code for error handling. Note that the same code will work, without any modification, irrespectively of the number of drones in each team. Also, it is straightforward to extend the code so that the mission program invokes the *checkJoin*() operation in a more intelligent way, e.g., when any of the two teams become empty (this is currently done only when both teams are empty), as well as to invoke the *forceLeave*() operation for nodes that are running out of fuel (in the current version, nodes will unilaterally sign-off and stop responding, thus they will be declared as failed).

We have performed extensive tests to verify the functionality of TeCoLa and its ability to support non-trivial mission programs, like the one above. For pactical reasons, these tests are conducted using a simulated setup, which includes virtual drones that run the TeCoLa node stack including the services that are provided by each node, and a virtual control station that runs the TeCoLa mission controller stack and the mission program. The details of the simulator environment are described in Chapter 7. It is important to note that the full TeCoLa stack can be transferred to a real platform (that runs an embedded version of Linux with the required devices drivers and Python runtime/libraries) with very few modifications.

In these simulated mission executions, nodes run dummy service implementations (except for the mobility service. More specifically, the fire detection service (provided by the the scanners) does not actually take/process images in order to report the location and the intensity of a potential fire outbreak. Instead, it is initialized with predefined locations for the fire outbreaks, so when a scanner flies above these locations and the respective detection service call is invoked, a fire spot along with its intensity is returned to the mission program (in all other cases, these invocations return an empty result). Along the same lines, the fire extinguisher service (provided by the fire extinguisher drones) implements dummy service

calls for the spray and load operations. For simplicity, a spray operation is assumed to completely eliminate the fire in the area where the water drop is performed, but the service implementation can be extended easily so that this happens after a series of attempts. It is important to note that all nodes run a real implementation of the mobility service, which can be used in real UAVs without modifications.

3.5 Performance evaluation of remote operations

The main performance overhead of TeCoLa comes from the remote operations between the mission controller and the nodes. Due to the communication that has to be performed over the wireless network, remote operations are significantly slower than any operations that are performed locally within the mission controller runtime environment.

There are three remote operations: (i) the join operation used to add nodes to the mission group; (ii) the leave operation used to remove operational nodes from the mission group (nodes that fail are automatically removed from the mission group); (iii) the remote calls used to invoke the services of the nodes that are part of the mission group. The join/leave operations are implemented using the corresponding primitives of GCBRR without any significant additional processing at the level of the TeCoLa runtime, and their performance is discussed in Chapter 5. The remote call operation is implemented using a transport primitive of GCBRR for conducting a request/reply interaction between 1 caller (in our case, the mission controller runtime) and N callees (in our case, the node runtimes). We refer to this interaction as a 1-N request/reply. The performance of 1-N request/reply interactions at the level of GCBRR is discussed in Chapter 5. In the following, we analyze and experimentally evaluate the delay of service calls at the level of TeCoLa, for different prototypical cases which differ in terms of the size of the application-level payloads.

The delay of a basic 1-N request/reply interaction for a null service that does not perform any processing and merely sends back a reply, can be estimated analytically as

$$RR_{a/b/N} = 1 \times MsgT_a + N \times MsgT_b \tag{3.1}$$

where $MsgT_x$ is the time it takes to transmit a packet with payload size x. In words, the delay of a 1-N request/reply interaction it is equal to the time needed to transmit the request plus the time needed to transmit each of the replies. Note that the above assumes perfect communication without any packet loss, as it does not include any provision for timeouts and retransmissions. Also, for simplicity, the replies of all nodes are assumed to have the same size.

We have experimentally measured RR for indicative corner cases where the request and/or the replies are either empty or fully loaded. As a typical wireless network technology, we use WiFi. Full packets carry the maximum allowed payload (1500 bytes), while empty packets only have the protocol headers (35 bytes). The measurements are performed using the WiFi simulation support of NS3, as well as using the NITOS testbed [3]. We run
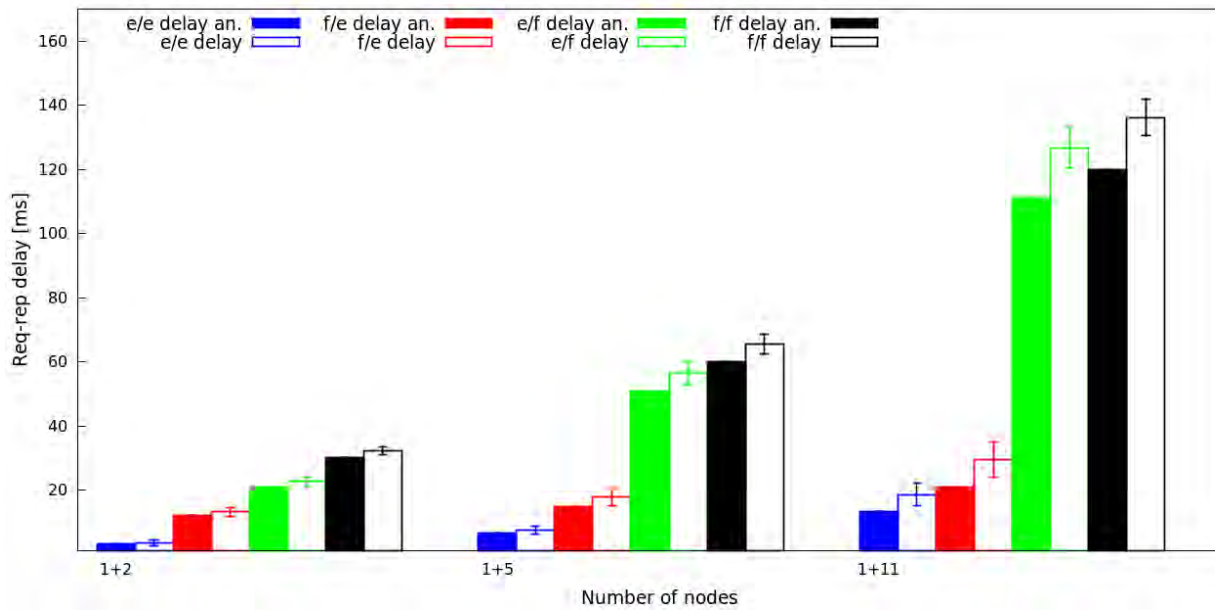
Figure 3.7: Delay for 1-N request/reply interactions with full (f) or empty (e) payloads.

WiFi at the basic rate of 1 Mbps, which yields a transmission time $MsgT_e$ of roughly 1 millisecond for empty packets, and a transmission time of $MsgT_f$ about 10 milliseconds for full packets. In both cases, the transmission time includes a constant overhead of about 0.7 milliseconds, which is introduced by the TeCoLa stack. The simulations produce virtually the same numbers as the real experiments, so here we only show the former.

Figure 3.7 reports the results for various 1-N interactions (where N is the number of nodes being invoked by the mission controller). We use the shorthand notation a/b to denote the size of the request (a) and the size of the replies (b), respectively. As an example, e/f stands for a request/reply interaction where the request is empty and all replies have a full payload, in which case the delay is analytically estimated as $RR_{e/f/N}$. We report the average delay over 1000 interactions (in all experiments, the standard deviation is small, on average less than 3%).

As can be seen, the delays grow linearly to the number of nodes. Also note that the analytically estimated values, which are derived using Equation 3.1, are very close to the experimental results. This is because all interactions are performed without any packet loss at the transport level (GCBRR avoids contention/collisions by design, see Chapter 5), so there is no additional delay due to timeouts and retransmissions (just as this is assumed in the equation).

The f/e interactions are representative for service invocations with parameters that fill the entire request packet but the replies are empty and serve merely as an acknowledgment of the call (typical for asynchronous service operations). In this case, the delay is relatively small, even for a larger number of nodes. This is particularly important for the scalability

of larger data transfer operations from the mission controller to the nodes, where the data does not fit into a single packet, which are internally implemented by TeCoLa as a sequence of basic 1-N request/reply interactions. The same holds if the parameters of a service call do not fit into a single packet, in which case the request is transferred to the nodes in several chuncks via consecutive 1-N request/reply interactions. The respective delay can be estimated as $RoundUp(Data/MaxPayload) \times RR_{f/e/N}$, where $Data$ is the amount of data to be transferred from the mission controller to the nodes, and $MaxPayload$ is the maximum payload size (1500 for WiFi).

The e/f interactions are representative for service calls that retrieve state information from all nodes, e.g., when polling conditions, where the request is virtually empty but the replies may carry a lot of data that fill-up an entire packet. In this case, the delay strongly depends on the number of nodes. In fact, the performance of e/f interactions approaches that of f/f when the number of nodes increases. This is because the delay for sending the full request is amortized from the large number of full replies. If nodes wish to send a reply that does not fit into a single packet, the mission controller runtime will repeat the 1-N requst/reply interraction as often as needed. In a similar vein as above, the respective delay can be estimated as $RoundUp(Data/MaxPayload) \times RR_{e/f/N}$, but in this case $Data$ is the amount of data to be transferred from the nodes to the mission controller.

Note that in the general case a 1-N request/reply interaction at the level of TeCoLa may take several f/e interactions for the transfer of the call parameters to the nodes, as well as several e/f interactions for the transfer of the replies back to the mission controller. The delay would then be equal to $RoundUp(Request/MaxPayload) \times RR_{f/e/N} + RoundUp(Reply/MaxPayload) \times RR_{e/f/N}$, where $Request$ is the size of the request including the call parameters, $Reply$ is the size of the replies/results, and $MaxPayload$ is the maximum packet payload.

3.6 Related work

Recently, several programming abstractions and languages have been proposed to simplify the development of applications that employ mobile robots. In the following, we review indicative works, and compare them with TeCoLa approach.

The Robotic Operating System (ROS) [87] adopts a modular and loosely-coupled architecture for organizing the different functions of a robotic system into distinct components, which can implement a variety of operations, from the handling of hardware resources to path-planning and navigation algorithms. ROS components interact with each other by exchanging messages using a publish/subscribe mechanism. A component may also provide services that can be invoked in an explicit way, as in TeCoLa. The Mission Oriented Operating Suite (MOSS) [81] follows a similar approach to ROS. In this case, the individual functional components are implemented as separate processes, which communicate with each other indirectly, through a server that routes the messages being published to the respective subscribers. ROS and MOOS are mainly designed for single robots, and focus on decoupling

the different components of a robotic platform. Still, given their pub/sub architecture, they could be adapted to support distributed multi-robot systems where communication among nodes/processes takes place over a network. TeCoLa operates at a higher level, and could be implemented on top of such distributed pub/sub frameworks.

In URBI [28], a robot comprises a set of abstract device objects. Each device is associated with a sensor or actuator, and implements a set of services for accessing the particular resource. URBI follows a client/server architecture whereby the robot act as a server, and the client connects to it via telnet. The client then uploads and runs the application program, which is written as a script that accesses one or more devices of the robot, as needed. While the resource abstraction is similar to that of TeCoLa, resource access is a strictly local operation and does not occur over the network. Another major difference is that URBI only targets single robots, without providing any support for coordinated group/team operations.

Other frameworks such as [41] and [42] also follow a service-oriented approach. In this case, robotic resources and capabilities are captured in the form of web services [22], which can be invoked in a structured and transparent way from a remote machine. However, unlike TeCoLa, these frameworks are designed to operate with a fixed and well-defined group of robots and resources, and do not offer any advanced programming support for service/team dynamics.

The Miro [50] middleware employs an object-oriented architecture, structured in three layers. The first layer provides platform-specific service abstractions for the sensors and the actuators of the robot, the second layer exports these services via the CORBA interface definition language, and the top layer provides higher-level services such as navigation and path planning. The programmer accesses these services via the standard CORBA object protocols. Miro has similarities with TeCoLa in terms of the resource abstraction model, but uses a more complex and language-neutral middleware for service invocation. Also, Miro targets static robotic groups without allowing the dynamic addition of resources at runtime, and the burden of coordinating a robotic team is left entirely to the programmer.

Karma [47] is an application programming model targeting resource-limited micro-aerial vehicles that do not have any communication capability when operating in field. A pool of drones is stationed in the so-called hive, which has a central data store for keeping the information generated by the drones. The application logic consists of drone behaviors along with activation predicates and progress functions. The activation predicates are evaluated based on the information that is currently available in the data store, and in turn can lead to the activation of a behavior. The hive allocates one or more drones to execute the activated behavior, which then fly out to perform the assigned mission. When the drones return, they offload the data collected/produced to the data store, and the respective progress function is evaluated to assess application progress. Like TeCoLa, Karma adopts a high-level coordinated approach, but the coordination is rather coarse and cannot be adjusted in a flexible way—once a drone leaves the hive, it operates independently from other drones, and does not communicate with the data store during the course of a mission.

In Meld [25], an entire swarm of robots is programmed as a single entity, using a declarative language. The resources provided by each robot as well as the desired high-level

achievements of the swarm are represented through facts and a set of rules. The rules are evaluated progressively by the swarm to produce new facts, until the top-level achievement is satisfied. Rules are propagated to the entire swarm, whereas facts are distributed among the robots based on their capabilities. If a computation requires facts that are not locally available, it retrieves them from other robots in the neighborhood. Thanks to the declarative nature of Meld, the programmer can formulate relatively simple missions in an easy way. However, unlike in TeCoLa, one cannot control/steer robots in an explicit way, in order to customize the swarm behavior according to mission-specific needs or to support more dynamic missions.

The Proto language [27] has its roots in the parallel computing domain. The space where the robots operate is abstracted as a so-called amorphous medium. Each point of the medium is a computational device that is associated with a tuple of values, which are internally mapped to one or more mobile robots. Mission-level operations are performed on the medium, while a functional oriented compositional approach is used to transform space operations into instructions for the individual devices, and ultimately into commands for each mobile robot. Proto comes with support for multi-robot operation and coordination, but this is implicit and at the granularity of spatial neighborhoods. In contrast, TeCoLa supports more versatile team formation, based on services and can more purposefully exploit heterogeneity.

Voltron [79] targets active sensing missoins where system behavior is adapted according to the sensor data. Its key abstraction is that of a single virtual drone, which features the entire set of programming primitives. The mission program is written for this virtual drone, specifying a number of sensing operations that need to be performed at certain locations. The underlying system coordinates the individual drones that are available in order to perform these sensing operations, based on a virtual synchrony mechanism [53]. Voltron takes care of dynamic group management, and the programmer does not need to be aware of the number of available robots. However, the details of swarm scheduling and formation cannot be customized by the mission program. There is also no support for resource heterogeneity.

Buzz [86] is a language for programming heterogeneous swarms of robots. Similar to TeCoLa, the resources of each robot are accessed via method calls. In addition, it is possible to create swarms based on the resources found in the neighborhood of a robot, and the application can invoke both swarm-level and robot-level operations. One major difference is that Buzz follows a decentralized approach where swarm formation and control is performed in a peer-to-peer manner, whereas in TeCoLa this process is driven by the coordinator. Also, while Buzz allows for a robot to share selected state information with the swarm, this must be done in an explicit way using special broadcast-like operations (virtual stigmetry). Moreover, the entire mission program must be pre-installed on every robot, which makes it harder to employ robots on-demand, or to use the same robot for different or very dynamic application missions without reporgramming it.

TeCoLa combines some features of Voltron and Buzz, along with architectural elements of Miro and URBI in order to achieve an easy and versatile coordination of robotic groups where the resources may change dynamically based on the needs of dynamic missions.

Chapter 4

Tolerating mission controller failures

This chapter focuses on how TeCoLa tolerates fail-stop failures of the mission controller, allowing the mission program to resume its execution in a largely transparent way. We start by introducing the problems that arise when such failures occur, and state the desired fault-tolerance objectives based on a suitable notion of consistency that takes into account the particularities of cyber-physical systems. Then, we present the approach we employ in order to achieve these objectives and the extensions that have been made to TeCoLa for this purpose. We also provide an evaluation of our fault-tolerance mechanism. Finally we discuss the related work.

4.1   The problem

Consider a simplified version of the smart argiculture scenario presented in Section 2.1, where a single team of aerial drones equipped with sprayers fly over a crop field and visits problematic areas in which plants have contracted a disease, in order to spray some pesticide. This process is not as straightforward as it seems due to dynamics that must be taken into account by the mission program. For instance, local winds may lead to postponing the spraying action or to the re-positioning of the drones in order to increase spraying accuracy and effectiveness. A similar situation may arise due to the presence of animals or humans in the area.

It is also important to note that some of the operations that are carried out by the mission program impact the physical world in an irreversible way. This means that it is not possible to "undo" them, and in some cases it is not even acceptable to "redo" them. More concretely, once an area in the field has been sprayed it cannot be un-sprayed, and it is certainly not desirable to spray the same area more times than necessary.

These dynamics do not only complicate mission programming. They also make it hard to tolerate failures of the mission controller. The problem of tolerating failures in a way that allows the application to transparently resume its execution, has been addressed from the early days of distributed computing, and many notable solutions have been proposed [54, 90,

95]. The existing techniques and protocols strive to bring the system state as close as possible to the point of failure, or to a previous consistent state, while assuming that the application runs in a self-contained manner or without having a direct interaction with a non-digital external environment. However, in our case the execution of the mission program depends on the environment dynamics, and traditional recovery methods may leave the mission program with an information gap, provide the mission program with obsolete information, or re-execute the mission program in an inconsistent way, which, in turn, may lead to wrong decisions and undesirable actions.

4.2 Transparency requirements

In order to tolerate a failure of the mission controller in way that is transparent for the application programmer, two basic problems need to be addressed. Firstly, one must recover from the failure and restart the mission controller runtime. Secondly, the mission controller runtime must resume the execution of the mission program while maintaining –as far as possible– the logical consistency of the coordination. We capture the desired functionality in the form of three concrete requirements:

R1 Mission program restart. If the mission program fails, it will be eventually restarted, on the same or another mission controller, so that it can continue its execution.

R2 Consistency for critical operations. If the mission program has already performed critical actuation operations before the failure occurred, the mission program should resume its execution following-up on such operations, without re-executing them.

R3 Information freshness. When the mission program resumes, it should take actuation decisions based on up-to-date information, provided this does not violate the above requirement R2.

Our approach for achieving these requirements is described in the sequel. We assume that the mission program is deterministic, in the sense that all crucial actuation decisions are taken based on information that is retrieved from the nodes via service calls.

4.3 Approach

The usual way to satisfy requirement R1 is to take checkpoints, i.e., to save the execution state of the mission program so that this can be recovered after a failure. The saved state can be kept locally on the host of the mission controller runtime referred to as the *primary* mission controller, but also copied on one or more so-called *backup* mission controllers, so that the application can be restarted on another mission controller if needed. In our system model, shown in Figure 4.1, depending on the system configuration, the role of the backup
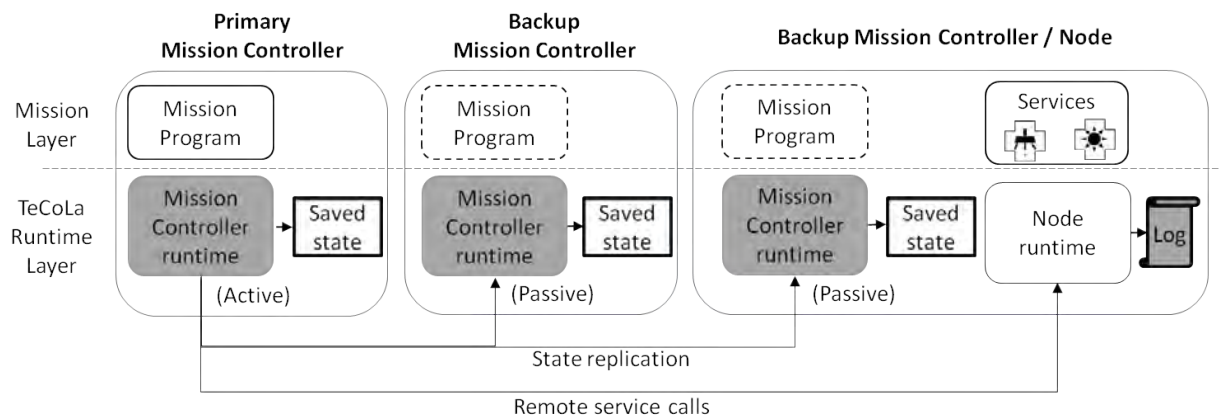
Figure 4.1: System model for tolerating failures of the mission controller.

mission controller can be assigned to a stand-by base station or a node (UV) with sufficient computing/communication resources.

However, a simple roll-back to the last checkpoint can violate requirement R2. On the one hand, since the decisions of the mission program are based on dynamic information that is provided by the nodes, the re-execution might follow a different path that is misaligned or even in conflict to critical operations that were performed in the context of the previous execution (before the failure occurred). On the other hand, if the re-execution follows the same path as the previous execution, this will lead to the re-invocation of operations that were already executed in the context of the previous execution. This is problematic if these operations have a cumulative, non-idempotent effect on the internal state of the node or on the state of the physical environment. To be consistent, as per requirement R2, the re-execution of the mission program should follow the same path as the previous execution, yet without the nodes repeating critical operations that were already performed in the previous execution.

To achieve the desired consistency, we introduce a form of logging on the side of the nodes. More concretely, nodes keep a log where they record the requests they receive from the mission program/controller along with the corresponding replies that were returned to it. When a node receives a request, it checks the log to see if it has already executed that service call. If the call is found in the log, the node responds with the logged reply without re-executing it. This ensures that during re-execution the mission program will receive the same information from the nodes as in the previous execution, and thus (assuming it is deterministic) it will follow the same execution path. It also ensures that previously executed calls, including the ones that correspond to non-idempotent operations, will not be re-executed by the nodes. When nodes reach the end of their logs, they start executing service calls as usual.

While the above combination of checkpointing and logging guarantees requirement R2, it does not achieve requirement R3. This is because the mission program will receive logged

information from the nodes, even when no critical operations have been executed in the previous execution. From a consistency perspective, in this case, it is better to let the nodes ignore their log and simply re-execute all incoming calls in order to return fresh information to the application.

To achieve the desired functionality, during the re-execution of the mission program, we use the node's logs selectively, only as long as this is needed to satisfy R2, and stop as soon as R3 can be satisfied. We refer to this approach as *selective replay*. In order to support the proposed approach, we extend TeCoLa in two ways.

Firstly, the mission controller runtime environment is extended so that it can record its state. This includes the execution state of the mission program as well as the mission group and the teams that have been created to this point. Also, this state information is replicated to all backups of the mission controller environment.

Secondly, to guide the replay of the mission program, we introduce the notion of *failure-persistent* (or persistent) service calls. When recovering from a failure, the mission program is re-executed in replay mode, with nodes responding to the calls received with the logged replies. Replay ends when the mission program reaches the point where the last persistent call was performed in the context of the previous execution (or the log's end is reached).

Whether a method is considered to be failure-persistent depends on the effects it has on the internal state of the node and the physical environment; it may also depend on the way each service implements the fail-safe state. For instance, calls that merely return state information are naturally non-persistent; in fact, as per requirement R3, it is desirable to re-execute them after a recovery, provided these were not succeeded by a persistent call in the previous execution. The node runtime knows which service calls are failure-persistent based on a special attribute of the respective interfaces, and informs the mission controller runtime accordingly.

As before, if a node detects a failure of the mission controller, it enters in a fail-safe state. More specifically, each local service is notified so that it can take the necessary fail-safe operations. Recall that remote service calls in TeCoLa always run to completion, thus some asynchronous/long-running operations may be kept active despite the failure, whereas others may be aborted when entering the fail-safe state.

4.4 Mission program replay

In the following, we describe in more detail how mission program replay works. Figure 4.2 (a) illustrates the high-level flow of this process from the perspective of the mission controller runtime while Figure 4.2 (b) from the perspective of the node runtime.

As mentioned above, if the mission program/controller fails, nodes enter a fail-safe state. When the mission controller runtime recovers (on one of the replicas), it retrieves the mission program state from the last saved checkpoint and enters the replay mode. It then asks all nodes to set the replay pointer to the beginning of their log and to indicate whether the log contains any persistent service call. If no node has a persistent call in its log, the mission
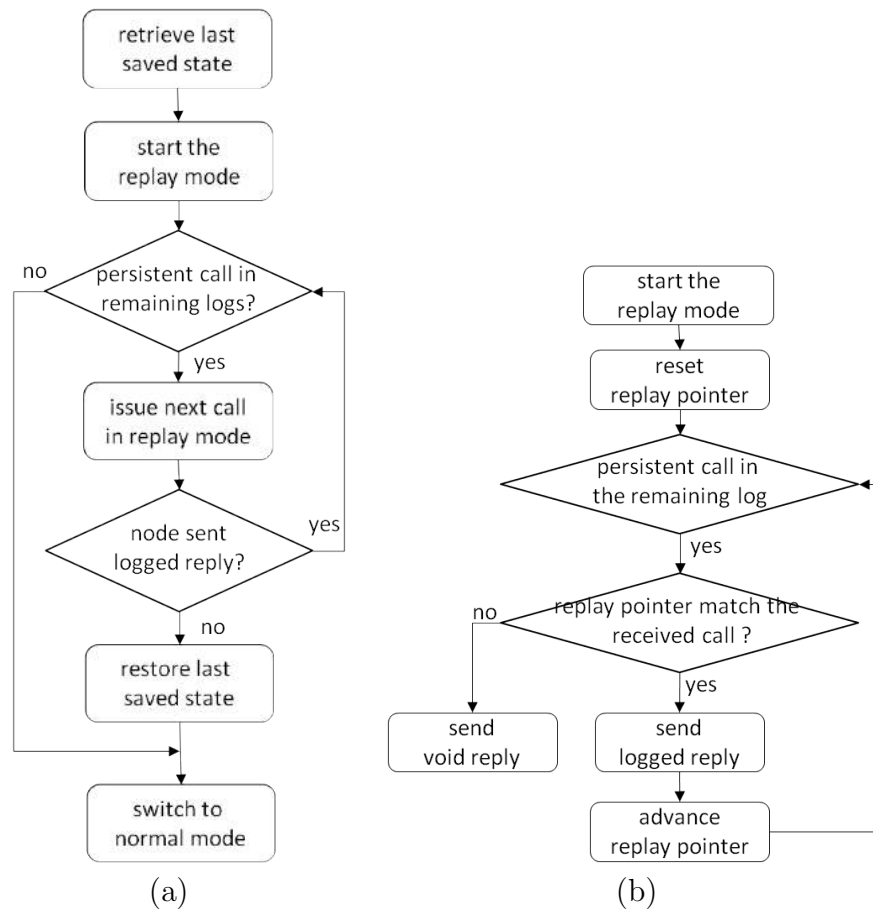
Figure 4.2: Flow diagram of the replay phase (a) for the mission controller runtime, and (b) for the node runtime.

controller runtime switches to normal execution mode (and informs the nodes accordingly), else it resumes the execution of the mission program in replay mode.

For every call issued in replay mode, the node inspects its log at the position of the replay pointer. If it finds the request in the log, it replies with the logged reply without re-executing the call, and moves the replay position forward in the log. If the node does not find the request in the log, it sends a void reply indicating that the call is not found. In any case, the node's reply carries additional information indicating whether the end of the log has been reached or whether a persistent call exists in the remaining part of the log.

If the mission controller runtime receives a logged reply from the node, this is returned to the mission program. Also, if this call was the last logged persistent call (there is no node with a persistent call remaining in the rest of its log), the execution of the mission program is switched to normal mode so that all nodes will execute subsequent calls as usual. Else, the execution of the mission program continues in replay mode.

If the node sends back a void reply, this means that the re-execution did not follow

the same path as the previous execution, and that the mission program behaved in a non-deterministic way[1]. In this case, it is not possible to achieve the desired transparency, so the mission controller runtime rolls-back to the last saved checkpoint, switches to normal operation, and notifies the mission program via an exception to handle the situation.

4.5 Resetting the log

During replay, a node merely advances the replay position each time it responds with a logged reply, but it does not delete any entries from the log. This way the log can be reused for a new replay, in case the mission program experiences yet another failure during replay or before taking the next checkpoint.

The logs of the nodes are reset/cleared when the next (logging) epoch starts as shown in the high-level flow process of Figure 4.3. The current logging epoch is identified using a sequence number, which is included to the service call requests sent to the nodes. When the next checkpoint is taken, the mission controller runtime increases the epoch number, and continues with mission program execution as usual. When a node receives a call request, it checks the epoch number with its own. If these are equal, it processes the request as usual, else the node increases its own epoch number, deletes all log entries that belong to the previous epoch, and processes the request as usual.

Note that a failure of the mission controller may occur after taking the next checkpoint but before or while making the next service call. This is not a problem. Upon restart, the mission controller runtime will enter the replay mode in the context of the most recent checkpoint, and it will ask the nodes to check their logs for persistent calls for the current epoch. If a node has already made the transition to the next epoch (then it has received/executed a service call after the mission program took the last checkpoint), it will reply as usual. Else, the node will increase its epoch number, clear the log, and reply that its log is empty. In any case, the mission controller runtime will proceed based on the replies received as discussed above.

Note that the size of the log could be reduced by letting the mission controller runtime automatically take a checkpoint right after each persistent call. However, as will be shown in the evaluation, checkpoints can be quite expensive and should not be taken too often without good reason.

---

[1]A node will send a void reply either because the call is not the next in the log, or because the end of the log has already been reached. It is clear that the first case directly identifies a non-deterministic re-execution of the mission program. The second case is less obvious. Recall that the mission controller runtime will send a call request to a node n_i in replay mode *only* if there is at least one node with a persistent call in its remaining log. Since n_i replies that its log is empty, the persistent call has to be in the log of another node, let n_j. The fact that n_j has a non-empty log (which also contains a persistent call) while the invoked node n_i has an empty log, means that in the re-execution the mission program invokes nodes in a different order than in the previous execution.
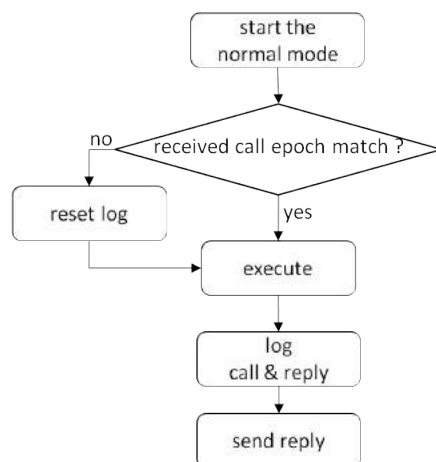
Figure 4.3: Flow diagram of log reset phase.

4.6   Extensions to the TeCoLa implementation

To implement the proposed fault-tolerance support, we extended the prototype TeCoLa implementation in several ways. In the sequel, we briefly discuss the most important ones.

Checkpoint operation

The checkpoint operation of the mission controller runtime is exposed to the programmer in the form of a new method of the *MissionController* object. The mission program can invoke this method to take checkpoints in an explicit way, whenever this is desired. Each checkpoint denotes a point of no return in the mission program flow: once taken, the mission program will never roll-back to an earlier state.

Internally, the state of the execution is recorded using the the DMTCP framework [24]. We use this out-of-the-box, without attempting an elaborate integration with the mission controller runtime environment. To reduce the size of checkpoints, we perform incremental checkponting using the HBICT module [4], which works seamlessly with DMTCP.

The checkpoint images are replicated to the mission controller backups over the network, in the spirit of a primary-backup scheme [36]. The image transfer is performed using the standard 1-N request/reply transport function of GCBRR. The backups of the mission controller runtime environment are specified through corresponding configuration options.

The backups keep the mission controler runtime in an inactive state. When a failure of the primary mission controler is detected, an election takes place to decide which backup will take over in order to resume the execution of the mission program. The time needed to elect the new primary is negligible (GCBRR employs a special election mechanism that makes it possible to take a decision based on information that is already available locally, without any additional communication, see Chapter 5 for more details). The elected replica then activates the mission controler runtime, which starts the mission program replay.

Polling conditions

When the mission progam waits on one or more conditions during normal execution via the *wait* method or the *select* method of the mission controller object, it is not known in advance how many times the mission controller runtime will actually invoke the service calls in the respective conditional expression before the condition becomes true. It could be that the condition is satisfied already after the first polling iteration, or not at all in which case the wait operation will return after the specified timeout and an exception will be raised. This is a problem during replay, given that the objective is for the re-execution to follow the same execution path/flow and to issue the same calls in the same order as in the previous execution.

To this end, when the mission controller runtime polls conditions in replay mode, it handles the replies received from the nodes in a slightly different way. In this particular case, it is acceptable for *all* nodes that are invoked when polling the conditional expression to reply that they cannot find the call in their log. This indicates that at this point the program stopped waiting (polling) in the previous execution, consequently the same is also done during replay.

It is important to note that the (potentially large) user-supplied timeout of the *wait/select* operations is not taken into account during replay, and thus does not introduce any extra delay. If a condition is satisfied during replay, the mission controller runtime knows that it was also satisfied in the previous execution, and the *wait/select* operation returns normally. Else, if all nodes reply that the call is not in their log, it is inferred that in the previous execution the *wait/select* operation returned due to a timeout, and such an exception is raised during replay too. We note that the mission controller runtime polls conditions in a deterministic order which can be faithfully re-produced during replay.

Node failures

The assumption of deterministic mission program execution can be invalidated if a node fails during normal execution. This is because the mission program does not control when a node will fail and when the corresponding exception will be raised. To enable a consistent replay, during normal execution the mission controler runtime automatically takes a checkpoint when a node failure occurs, before raising an exception. This way, in case of a failure, the mission program will resume from that point and will re-execute without being affected by non-deterministic external factors.

However, this only addresses half of the problem: namely, a node that was available in the previous execution may fail during replay. In this case, it is generally impossible to achieve the desired transparency, so only a best effort is made to continue with the replay. More specifically, if the mission controller runtime detects a node failure during replay, it records this internally but does not notify the mission program as long as it does not attempt to invoke a service call of the failed node. Depending on the mission program flow and how far the previous execution progressed since the last checkpoint, the replay may be completed

transparently, in which case the mission controller runtime will notify the mission program as soon as it switches to normal mode (after taking a checkpoint).

If the mission program invokes the failed node during replay, the mission controller runtime can no longer hide the failure and will raise an exception. But no checkpoint is taken in this case, and the execution of the mission program continues in replay mode. This is an optimistic approach, in hope that despite this disruption the mission program might still behave in the same way as in the previous execution. Of course, this may not happen and a deviation with respect to the previous execution might be detected during the continuation of the replay. Then, as already discussed, the mission controler runtime will roll-back to the last checkpoint and will notify the mission program so that it may handle this problem in an explicit way.

4.7 Analytical cost estimation

This section presents an analysis of the overheads introduced by our fault-tolerance mechanism, as this is currently implemented in TeCoLa. More concretely, the main overhead components of the fault-tolerance support are: (i) the checkpointing operation, including the state transfer to the backup replicas of the mission controller runtime; (ii) the recovery of the mission controller runtime, incuding the retrieval of the saved execution state in order to resume the mission in a smooth way; (iii) the mission program replay, including all service calls that are performed before the mission controller runtime switches to normal execution. We assume that the hardware platforms running the mission controller runtime and the node runtimes have sufficient memory resources, thus we focus on the delay aspect of the above overhead components.

Checkpointing

The delay of the checkpoint operation can be expressed as $CheckpointT = RecT + CopyT$, where $RecT$ is the time it takes for the mission controller runtime environment to record the state of the mission program and mission group, and $CopyT$ is the time required to copy/transfer over the network the image to all the (passive) replicas of the mission controller runtime that act as backups.

Recall that in our current implementation the execution state is recorded using the DMTCP framework [24] in conjunction with the HBICT module [4]. Given that checkpoints are taken incrementally, $RecT$ basically depends on the number and size of data objects that were created/modified by the application since the last checkpoint. A concrete example is given in the next section.

The delay $CopyT$ for the image transfer over the network to the backup replicas of the mission controller, can be estimated as $RoundUp(Image/MaxPayload) \times RR_{f/e/N}$, where $Image$ is the size of the checkpoint image, $MaxPayload$ is the maximum packet payload, and $RR_{f/e/N}$ is the delay for a 1-N request/reply interaction with a full request and empty replies,
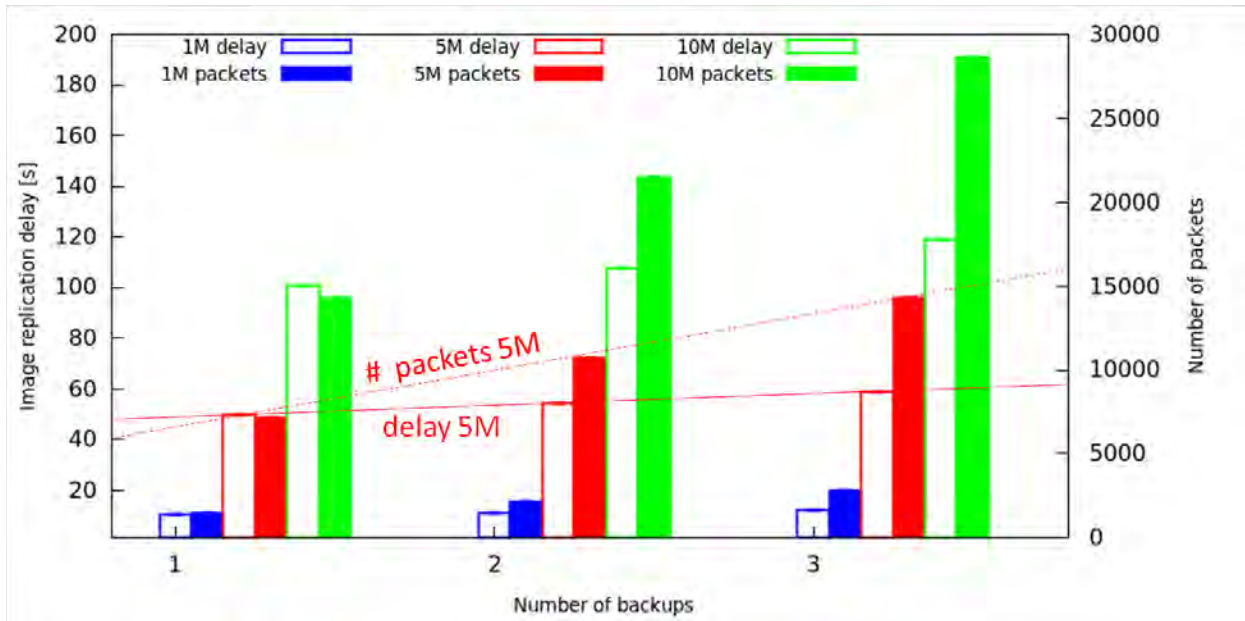
Figure 4.4: Image replication delay (left y-axis) and number of transmissions (left y-axis).

according to Equation 3.1 given in Section 3.5. We have measured $CopyT$ for the same network configuration that was used in the performance experiments in Section 3.5 (WiFi at 1Mpbs, where the empty/full packets are 35/1500 bytes long and have a transmission time of about 1 and 10 millseconds, respectively). Here too, the measurements are performed using both the WiFi simulation support of NS3 and the NITOS testbed [3].

Figure 4.4 reports the values for typical checkpoint images of 1M, 5M and 10M as a function of the number of backup replicas of the mission controller runtime. The real measurements as well as the analytically calculated values are again very close to the measurements obained via NS3, and are not shown to avoid clutter. The results show a stronger dependence on the the size of the checkpoint image rather than on the number of backups. For example, as the red line indicates, the time it takes to transfer a 5M image to 1 backup is almost the same as for 3 backups, despite the fact that the number of transmissions grows linearly to the number of replicas as shown by the red dotted line. This is thanks to the scalability of f/e 1-N request/reply interactions (see Section 3.5). Similar observations can be made for the other image sizes.

Recovery

The mission program recovery delay can be expressed as $RecovT = DetectT + StartT$, where $DetectT$ is the time required to detect the failure of the old primary mission controller, while $StartT$ is the time needed by the new primary in order to extract the execution state from the most recent image and to restart the mission program execution.

DetectT basically depends on the programmable aliveness timeout (see Section 3.3), which is typically set generously so that the mission controller runtime does not have to issue heartbeats too frequently Once the failure of the mission controller is detected, the time required to elect a new primary is negligible. This is because GCBRR employs a special election approach, which allows all replicas to determine the next primary without any communication between them (for more details, see Chapter 5).

StartT depends on the size of the checkpoint image and the degree to which the mission program employs complex system-level mechanisms, such as mutli-threading and inter-process communication. A concrete example is given in the next section.

Replay

A rough estimate for the delay of the replay phase of the mission program can be calculated as $ReplayT = L \times RR_{f/f/N}$, where L is the log position of the last persistent call over all nodes. $RR_{f/f/N}$ is the delay of an f/f 1-N request/reply interaction as per Equation 3.1, which corresponds to a replayed serice call towards all N nodes of the mission group, and where both the request packet and the reply packes are fully loaded. The assumption is that every replayed call takes a single 1-N request/reply interaction between the mission controller and the nodes, which is quite realistic for most applications.

For the sake of completeness, we note that the above estimation does not capture the worst case. As mentioned in Section 3.5, a service call may take several f/e interactions to transfer the call parameters to the nodes, as well as several e/f interactions for the transfer of the replies back to the mission controller. This also applies to a replayed call, in which case the delay may increase depending on the size of the call parameters and replies. But this is unlikely to happen very often in practice.

4.8 Application case study

To put the above costs into perspective and to see how the overheads reported above affect the execution and recovery of a concrete mission, we use an indicative program for an application inspired from [52], where a team of 3 quadcopters is used to spray pesticides on crops. We assume a large crop field that is divided into sections, each containing a number of problematic areas that need to be sprayed. The spray operation is performed by all 3 quadcopters of the team when the wind conditions are suitable in terms of speed and direction.

Listing 4.1 shows a code excerpt of this mission program, in TeCoLa (without the team formation part, which works along the lines of the examples that are given in other parts of the thesis). Once the spraying team reaches the start of a section (line 6), a checkpoint is taken (line 8) to ensure that the mission will not roll-back in a previous section in case of a failure. Then, the team is instructed to visit the problematic areas of this section, one by one. When the team reaches the next target area (line 17), the current wind conditions are retrieved by invoking the (team-level) weather service of the drones (line 19). If these are

favorable, the team is instructed to perform the spray operation, and the area is removed from the list. Else, the mission program decides not to spray the target area, and moves the team to the next one. The section completes when all problematic areas have been sprayed. In this example, the spray operation is persistent: on the one hand, it is not desirable to spray the same area several times due to an mission prorgam restart; on the other hand, if the mission program restarts but the area has not been sprayed yet, it is desirable to re-check the wind conditions before taking the decision to spray.

Listing 4.1: Crop spraying application.

```
1  # team already created , as usual
2
3  for sect_init_wp in crop[sect_id]:
4   team.MobilitySvc.relativegoto(sect_init_wp)
5   cond = [[team.Mobility.getRelDistanceFromTarget,"<",1.0]]
6   mc.wait(cond,10,200)
7
8   mc.checkpoint()
9
10  spots = getProblematicSpots(sect_id)
11  spots_iter = itertools.cycle(spots)
12
13  while len(spots) > 0:
14   wp = next(spots_iter)
15   team.MobilitySvc.relativegoto(wp)
16   cond = [[team.Mobility.getRelDistanceFromTarget,"<",1.0]]
17   mc.wait(cond,10,200)
18
19   wind = teams.Weather.getWindCnd()
20
21   if checkWind(wind.speed,wind.direction):
22    team.Spray.spray(PESTICIDE_AMOUNT/team.size())
23    spots.remove(wp)
24    spots_iter = itertools.cycle(spots)
```

We run the mission program on top of a simulated setup consisting of three virtual drones and two virtual control stations (one being the primary and the other serving as a backup). The mission control TeCoLa stack is installed in both control stations, with the primary one serving as the active mission controller. The node TeCoLa stack is installed in each of the virtual drones that participate in the mission. The communication between all entities takes place over a simulated WiFi network that is implemented using the NS3 network simulator. The details of the simulator environment are described in Chapter 7.

Figure 4.5 shows a screenshot of the mission program execution, where the initial position of the team of three UAVs is at point A and the problematic spots that needs to be sprayed
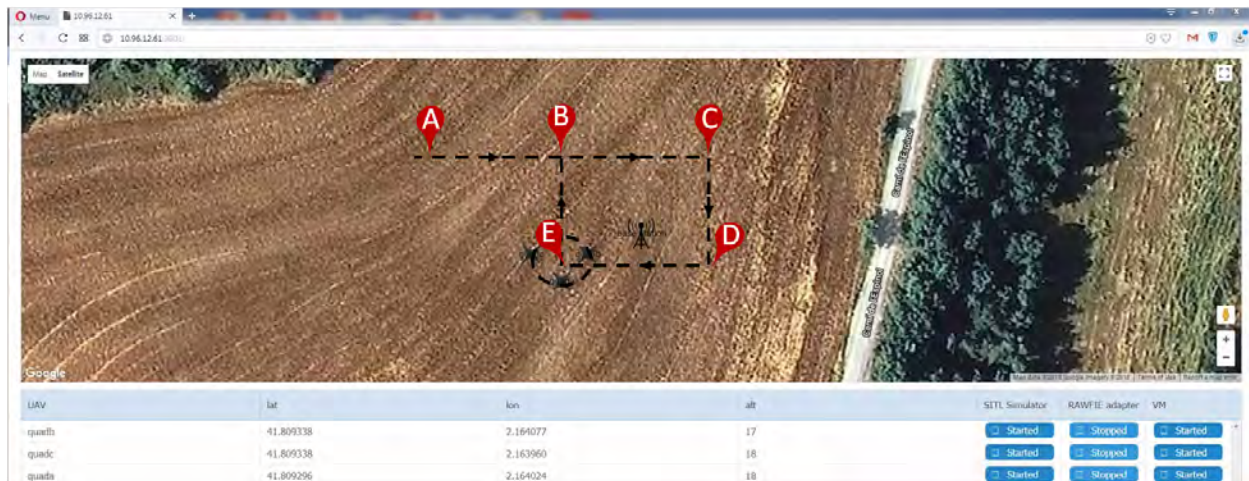
Figure 4.5: Visual snapshot of the spray application in the simulation environment.

are the points B,C,D,E. The start of the section is at point B. The waypoints are 60 meters away from each other, and the UAVs are flying at a speed of 1m/s.

Checkpointing overhead

In a first experiment, we let the application perform a full circle, visiting and spraying the spots B,C,D,E without any failure. We compare the execution of the application for two different cases, with the fault-tolerance mechanism being disabled and enabled, respectively. Figure 4.6 plots the recorded execution delays and the number of packet transmissions that are performed over the WiFi interface, for the parts of the mission code that are relevant for the fault-tolerance aspect. As can be seen, apart from the checkpointing operation, our mechanism does not pose any overhead to the execution of the mission program.

Each checkpoint operation takes 150 seconds. The time $RecT$ it takes to save the execution state is about 5 seconds. The rest of the delay, about 145 seconds, is due to $CopyT$, the time that is required to transfer the image from the primary to the backup control station. The image size is about 14Mb and the data transfer requires about 10.000 request/reply interactions between the primary and the backup control station (each requiring a transmission of 1 full request/data packet and 1 empty reply/acknowledgment packet, resulting in a total of 20.000 packet transmissions). Even though the checkpointing delay is significant in absolute terms, the order of magnitude is comparable to the time the mission prorgam passively waits (without doing any actuation or decision making) for the team to reach a certain position; in our scenario this waiting time is about 60 seconds, and it could easily grow if the team would have to visit more distant waypoints or would travel at slower speeds.

We note that the latency of the checkpoint operation could be reduced by compressing the checkpoint image, or it could be hidden by overlapping the image transfer with the execution of the mission program. Also, instead of using WiFi at the basic transmission
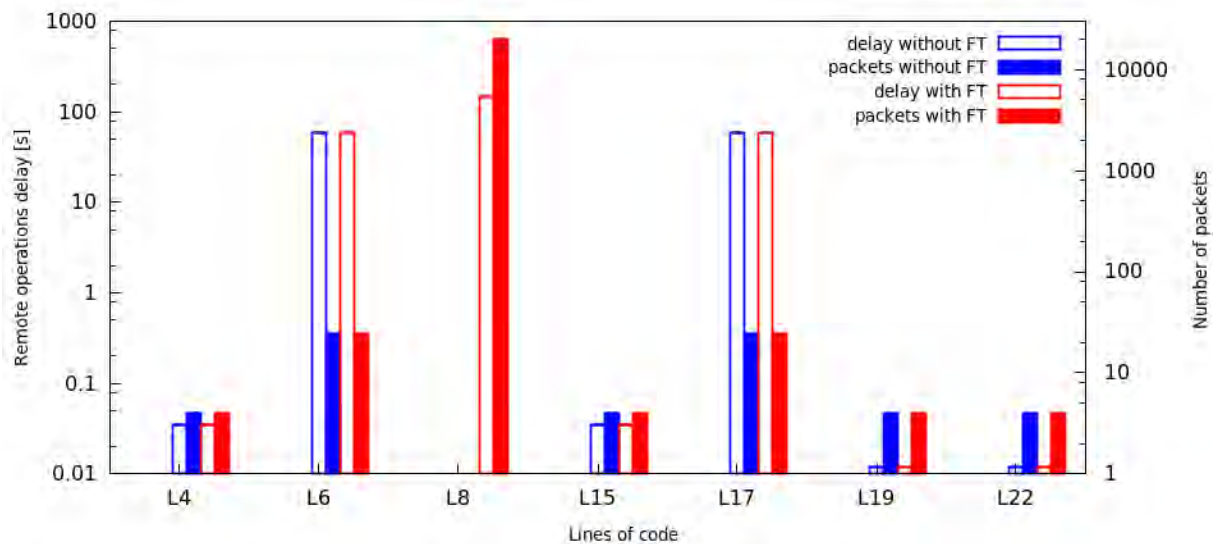
Figure 4.6: Execution time (left y-axis) and packet transmissions (right y-axis) per operation performed by the application for a failure-free execution. Both y-axes are in log scale.

rate of 1Mbps for broadcasting as we do in our experiments, one could employ the pseudo-broadcast transmission approach of [65] to exploit the highest possible rate of the wireless interface of 54 Mbps (which would reduce the image transmission time by more than 95%). Moreover, depending on the system configuration, the primary and the backup mission controller could communicate with each other over a faster and perhaps dedicated channel. Such optimizations are beyond the scope of this work. In any case, is clear that checkpoints are not for free, and should not be taken casually.

Recovery overhead

We perform a second set of experiments to show the overhead of the basic recovery steps in terms of latency and packet transmissions, for two different failure scenarios of the mission controller. In both cases, the team visits spots B,C but we manipulate the wind conditions so that the mission program decides not to spray these spots, and then let the team proceed to spot D where the mission program takes the decision to spray. In the first scenario, we introduce a failure right *before* the mission program invokes the spray operation, whereas in the second scenario we introduce the failure right *after* this invocation.

Figure 4.7 shows the main components of the recovery delay: (i) the time it takes to detect the failure of the primary mission controller ($DetectT$), (ii) the time it takes to resume execution of the mission program in the backup mission controller ($StartT$), (iii) and the duration of the replay phase ($ReplayT$). The cost of the first two components is the same for both failure scenarios. As already mentioned, the failure of the primary mission controller runtime environment is detected based on a configurable aliveness timeout. In our scenarios,
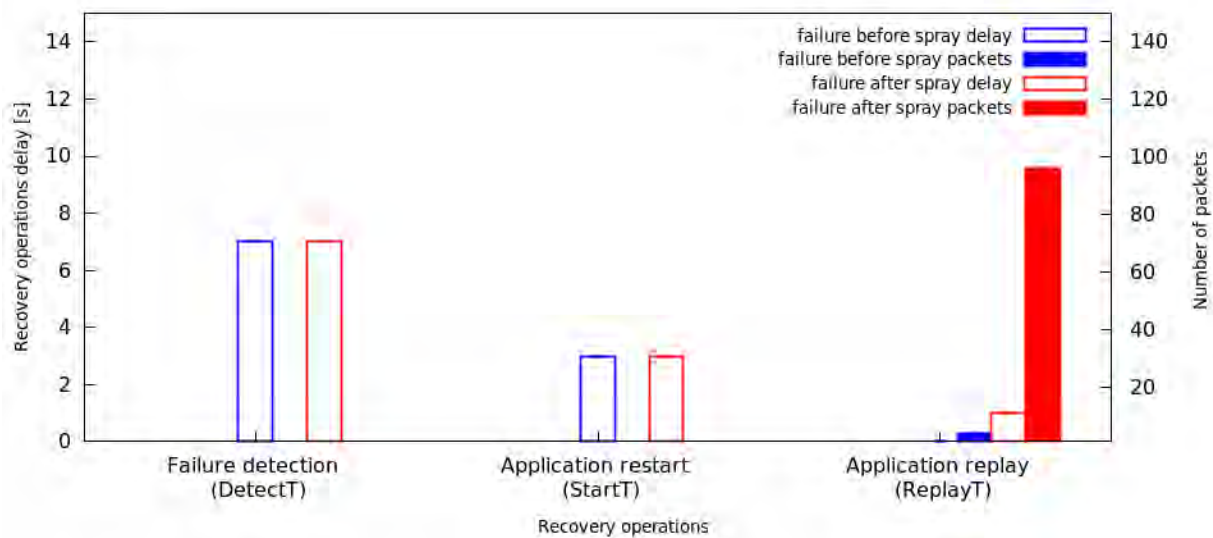
Figure 4.7: Delay (left y-axis) and number of transmissions (right y-axis) for the recovery operations that are performed when the application fails before and after the spray operation.

this is set to about 7 seconds, which corresponds to 3 consecutive missing heartbeats for a heartbeat period of 2 seconds, plus some tolerance (other missions might use more/less frequent heartbeats and smaller/larger timeouts). Thus $DetectT$ is roughly 7 seconds. The time $StartT$ to restart the mission program from the last checkpoint depends on the size of the checkpoint image and the system-level mechanisms used by the mission controller. For this particular application, this is roughly 3 seconds.

The replay delay $ReplayT$ is practically zero in the first scenario. Since the mission program fails before the invocation of the persistent spray operation, normal execution of the mission program resumes immediately after the last checkpoint without executing any calls in replay mode. The only overhead is due to the initial inquiry of the mission controller runtime about the existence of persistent calls in the nodes' logs. In the second failure scenario, the replay follows-up the previous mission program execution up to the point where the spray operation is reached. This includes 24 team-level service calls in replay mode, which are performed very fast, within 1 second. The actual recorded replay delay is 816 miliseconds. This is close but less than the analytical estimate of $L \times RR_{f/f/N}$, which amounts to 960 milliseconds for $L = 24$, $MsgT_f = 10$ milliseconds and $N = 3$. The reason for this difference is that the request/replies of the replayed serice calls do not occupy full packets. In any case, it is pretty clear that the replay is quite cheap, and nicely complements checkpointing which is much more costly and thus should be applied with care to avoid blocking the mission program during normal execution.

4.9 Related work

The problem of surviving failures is well-studied for traditional distributed computing systems. Seminal work includes distributed checkpointing and/or logging to record the internal system state and the information resulting from the communication with external systems, in order to recover the application in a consistent state after a failure [23, 39, 93]. In these approaches, recovery always follows the execution path of the application until the point of failure, based on the state information that was recorded during execution. Note that messages that have been sent towards an external system before the failure occurred, may be re-sent as a side-effect of recovery, so the external system must able to properly identify and handle such duplicates. In our work, the state of the mission programm is not distributed but resides centrally at the mission controller, so checkpoints can be taken locally without needing to involve other entities using a distributed protocol. Nodes correspond to external systems, where logging is employed to identify any duplicate requests that could be generated due to roll-backs. Going beyond conventional roll-back schemes, we implement a *selective replay* of the mission program in order to achieve the desired consistency at the application level.

A well-known approach for tolerating failures in mission-critical and hard real-time systems, is active replication [61]. In this case, different replicas are created for the system component of interest, where each replica is fed with the same input, actively processes it, and produces its own output. These outputs are then multiplexed by special arbitration logic that produces the final output, and possibly forwards it to the next component in the system pipeline. Besides being resource-intensive, such techniques typically require extensive communication and synchronization among the replicas to ensure that all replicas will execute in the same way –the usual assumption is that replicas receive their input via a total-order multicast [48]. In our case, such a synchronization among the replicas of the mission control runtime environment and the active execution of the mission program by all replicas would increase not only the traffic but also the on contention over the wireless medium, leading to collisions and retransmissions. Nodes would also have to include suitable arbitration logic in order to properly intercept and handle service calls coming from the different replicas. Since our work does not focus on tolerating byzantine failures, we adopt a passive replication approach [36]. The active mission control runtime acting as the primary replica, while the passive mission control runtimes acting as the backup replicas. Although the replication of the state information from the primary to the backup replicas is quite costly, the applicatios we target in our work can typically live with such delays. Also, with the exception of changes occuring to the mission group, checkpoints are only taken at the request of the mission program, knowing that this operation may take some time to complete.

Fault-tolerance approaches have also been studied for robotic systems. For instance, ALLIANCE [83] provides a fully distributed framework that allows groups of robots to collaborate in order to achieve a high-level goal. It employs a distributed artificial intelligent approach whereby each robot encodes the high-level mission goals along with the low-level controls into a hierarchy of mathematically modeled behaviors. The robots collaborate with

each other when needed according to their behaviors, and are able to detect whether the collective action is effective through a number of suitable sensors. If a robot that performs a collaborative task fails, other robots handle the failure by performing some corrective actions, based on locally defined behaviors that may depend on the robot's type and resources. Unlike in our work, the handling of faults is the responsibility of the application developer. Furthermore, for each application, the behavioral model must be designed to work in a distributed manner, and the respective parts must be pre-installed in the respective robots.

An architecture similar to ALLIANCE which allows robots to be used from a number of box-pushing applications (where the robots push boxes) without any re-programming is presented in [59]. In this case, the application is composed as a bundle of tasks that is loaded in a central task allocation system responsible for assigning tasks to available robots. Task assignment is performed through an auction-based scheme, taking into account the resource availability of the robots. Tasks can communicate with each other either locally or remotely according to the mission goals. When a failure is detected, the task allocator re-assigns the tasks of the failed robot to other robots. However, tasks are restarted and the programmer is responsible for handling any inconsistencies in the application state due to the restart. In addition, it is assumed that the central task allocation system (which basically corresponds to the mission control runtime in our system model) never fails.

A notably different programming framework for collaborative robotics with support for fault-tolerance is presented in [29]. The application logic is written in a functional way using the ML5 programming language [80]. The application is distributed among a group of robots, and its execution is performed through consecutive RPCs. The framework provides a primitive for the programmer to annotate critical blocks of code that might be problematic in case of a failure. Within these blocks, the application programmer defines a number of compensating actions that need to be performed if a failure occurs during execution. This ensures that the application can progress despite failures, e.g., the compensating actions can be used to avoid deadlocks, but the critical part of failure handling (the compensating actions themselves) remains the responsibility of the programmer. In our case, the application is centralized so no such issues arise, and robot failures are handled in a conventional way through exceptions in the context of service calls.

Chapter 5

Transport protocol

This chapter presents the GCBRR transport protocol, which is used to support the TeCoLa programming model. The main feature of GCBRR is its support for efficient coordinated 1-N request/reply interactions within a process group, and the management of the group in order to tolerate process failures as well as to allow processes to join and/or to leave the group. We start by discussing the motivation behind GCBRR. Then we present the protocol in detail, discuss its key properties, and informally argue about its correctness. Finally we compare the performance of GCBRR with different point-to-point approaches and we discuss the related work.

5.1   Motivation

As mentioned in previous chapters, the design of TeCoLa naturally fits the 1-N request/reply interaction pattern, since the mission controller may issue the same request towards several nodes, and each node node, in turn, sends a reply in response to such a request. Focusing on the salient characteristics of this interaction from the perspective of the transport layer, we abstract the mission controller and the nodes as a single process group, where the mission controller acts as the *coordinator* process that initiates such 1-N request/reply interactions and manages the process group, and nodes are *ordinary* processes that handle the coordinator's requests.

One way to support the 1-N request/reply communication pattern is to rely on unicast messaging, and implement the desired interaction via multiple point-to-point flows. The key disadvantage of this approach is that it leads to a large number of message transmissions, which increase rapidly to the number of processes. While this may not be a problem in very fast and reliable wired networks, it is highly undesirable in wireless systems where bandwidth is typically limited and messages can be lost due to collisions or delayed due to low-level flow control mechanisms used to avoid contention.

More specifically, when using unicasts one basically has two options. The first approach, illustrated in Figure 5.1 is for the coordinator to send the request via multiple consecutive
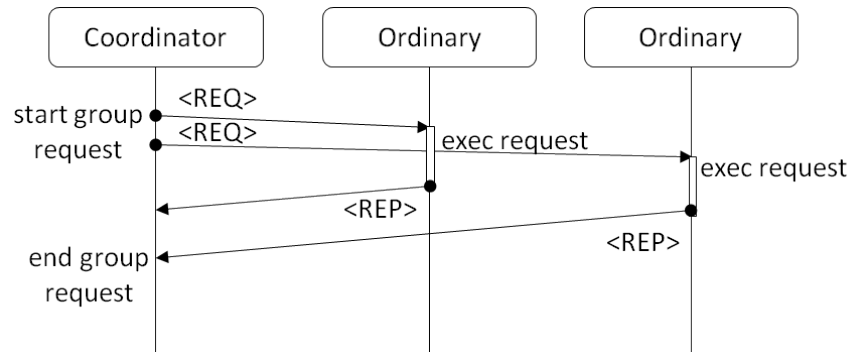
Figure 5.1: Message diagram for U-PAR.
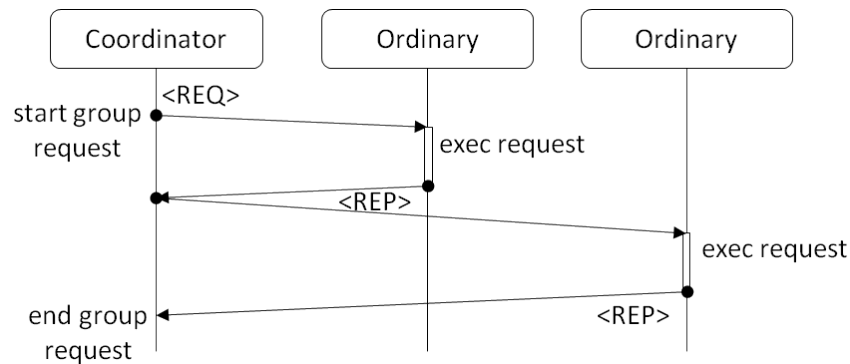
Figure 5.2: Message diagram for U-SEQ.

transmissions, each one targeting a different ordinary process, and then let each ordinary process send its reply as soon as this is ready. We refer to this as the parallel unicast protocol (U-PAR), because ordinary processes can handle the coordinator's request in parallel. However, this introduces the risk of contention since orindary processes might attempt to send back their reply at the same time. Also note that if requests are handled very fast, contention may even occur between the coordinator that sends the request and an ordinary process that sends back a reply. The second approach is for the coordinator to serialize the request/reply interaction with each ordinary process, as shown in Figure 5.2. We refer to this protocol as U-SEQ. While this avoids the contention problem, it comes at the cost of increased latency.

As an alternative approach, we propose a different protocol that does not use unicasts but instead exploits the physical broadcasting capability of the shared medium in order to (i) enable a fully parallel processing of the coordinator's request, (ii) reduce the number of message transmissions over the network, and (iii) avoid contention on the wireless channel. We refer to this protocol as *Group Coordinated Broadcast-based Request-Reply* (GCBRR). The basic idea of GCBRR is illustrated in Figure 5.3. The coordinator sends the request to all ordinary processes via a single broadcast. Each ordinary process sends back its reply
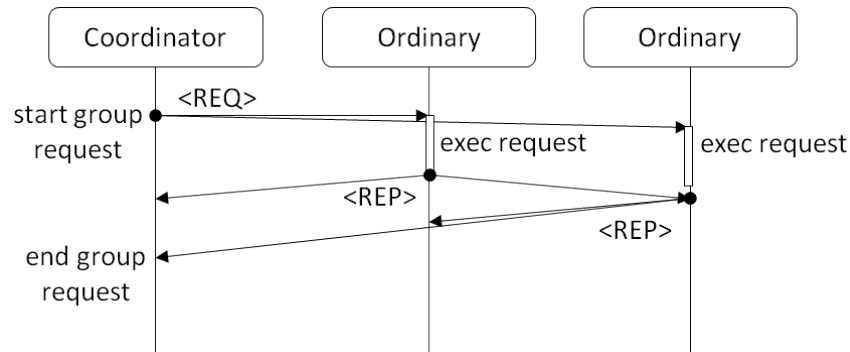
Figure 5.3: Message diagram for GCBRR.

Table 5.1: GCBRR protocol vs. unicast-based protocols.

Protocol	Transmissions	Latency	Contention
U-PAR	$2 \times (N-1)$	$ProcT + 2 \times (N-1) \times MsgT_x$	Yes
U-SEQ	$2 \times (N-1)$	$(N-1) \times (ProcT + 2 \times MsgT_x)$	No
GCBRR	N	$ProcT + N \times MsgT_x$	No

to the coordinator, again via a broadcast. The reply of an ordinary process is overheard by other ordinary processes, triggering the transmission of the next reply. The transmission order is specified by the coordinator as part of the request.

Table 5.1 provides a high-level comparison between GCBRR and the above unicast-based approaches, where N is the total number of processes in the group (coordinator and ordinary processes), $ProcT$ is the time it takes for a process to handle a request, and $MsgT_x$ is the time it takes to transmit a packet with payload size x (assuming equal latency for broadcast and unicast transmissions and equal size for the request and the reply payload). GCBRR reduces the number of message transmissions by $2x$ over U-PAR and U-SEQ. It even has $2x$ smaller aggregated messaging latency over U-PAR; note that the end-to-end latency also depends on the time $ProcT$ it takes for ordinary processes to handle a request (if $ProcT$ is large compared to $MsgT_x$ then this will dominate the latency of the interaction). Last but not least, like U-SEQ, GCBRR is by design contention-free, without relying on any MAC-level mechanism to avoid/detect collisions, or to throttle transmissions.

Notably, the above comparison does not take into account additional overhead that might be introduced by the lower layers of the network stack for unicast transmissions. For instance, the underlying MAC can use special signaling to deal with the hidden terminal problem, or it may employ its own acknowledgment and retransmission scheme in order to repair packet loss/corruption without involving higher layers. While such mechanisms are very useful to support independent point-to-point flows over wireless, they can translate to mere overhead for coordinated point-to-multipoint communication, which is the focus of GCBRR.

Table 5.2: Primitives of the GCBRR protocol.

Init(*pid*, *pids*)**:** Initializes the protocol with identifier *pid* for the local process. If *pids* is empty, the local process must explicitly join the group via the *Join* primitive. Else, *pids* contains the members of the group, including *pid*. By convention, the role of the coordinator is assigned to processes in increaasing order of their intentifiers.
RequestReply(*pids*, *data*, *procT*)**:** Sends to the set of destination processes *pids* a request with payload *data* and an estimated processing delay *procT*, and returns the set of replies produced.
ProcessRequest(*data*)**:** Process the next request with payload *data*, and return the reply.
Join()**:** Join the group.
CheckJoin(*joinT*)**:** Wait *jointT* for processes to join.
ForceLeave(*pids*)**:** Force a set of processes *pids* to leave the group.
BecameCoord(*pids*)**:** The local process has coordinator status, for the group view *pids* which includes all the processes of the group.
GroupChanged(*joined_pids*, *left_pids*, *failed_pids*)**:** The group view has changed due to processes that joined the group (the corresponding process ids are in the *joined_pids* set), and/or processes that left the group (*left_pids*), and/or processes that failed (*failed_pids*).

5.2 Protocol primitives and their usage in TeCoLa

Table 5.2 shows the programming interface through which the functionality of GCBRR is exposed to the upper software layers. The $Init()$ primitive is used for the initialization of the protocol at each process. The $RequestReply()$ primitive is used by the coordinator to perform a request-reply interaction with the ordinary processes, while the $ProcessRequest()$ handler is used by an ordinary process to process the next request of the coordinator and return the reply for it. The $Join()$ primitive is used by a process that wishes to join the group. In order for this to happen, the coordinator has to invite nodes to join via $CheckJoin()$. The coordinator can evict ordinary processes from the group via $ForceLeave()$. Note that an ordinary process has to follow the coordinator's command to leave the group. Finally, the $BecameCoord()$ event handler informs a process that it took over as the group coordinator, while $GroupChanged()$ is invoked at the coordinator process in order to handle group changes including process failures.

This API is used by the lower layers of the TeCoLa stack, the mission control runtime and the node runtime environments, to support the functionality that was discussed in the previous chapters. More specifically, $Init()$ is called with a non-empty process list only by the replicas of the mission control runtime. The primary is the one with the smaller process identifier, and backup replicas will take over as the new primary (if needed) in increasing order of their process ids. The node runtime invokes $Init()$ with an empty process list, and subsequently invokes $Join()$ in order to join the mission group. The $CheckJoin()$ and $ForceLeave()$ primitives are used by the mission control runtime to let nodes join and to force nodes leave the mission group, whenever this is requested by the mission program through the corresponding primitives of the $MissionControl$ object (see Chapter 3). Through the

GroupChanged() primitive, the mission control runtime updates the mission group and the teams that have been created by the mission program. The *BecameCoord* event leads to the activation of a backup replica of the mission control runtime in order to take over as the new primary. Finally, the *RequestReply*() and *ProcessRequest*() primitives are used to implement the node-level and team-level remote service calls, to confirm the aliveness of the mission controller to the nodes and to receive any service provision updates, to transfer checkpoint images from the primary mission control runtime to the backups, as well as to perform the application replay when recovering from a failure of the mission controller.

5.3 Protocol description

The GCBRR protocol supports the above primitives in a structured and modular way. It consists of the: (i) the basic *request-reply* protocol; (ii) the *join* protocol; (iii) the *leave* protocol; (iv) the *view-push* protocol; (v) the *fault handling and election* procedure. This approach makes it easier not only to describe but also to implement/test the protocol. Of course, these components are not entirely orthogonal to each other, but the interplay between them is limited. Also, it is straightforward to drop some of these components in case the upper layers do not require the full functionality. In the sequel, we discuss the system model and key assumptions, and present the individual protocol components.

System model and assumptions

Let there be N processes, with monotonically increasing identifiers, p_i, $1 \leq i \leq N$ where $p_i < p_j \iff i < j$. The processes share the same broadcast domain and communicate via broadcast or unicast transmissions. Both types of transmissions are unreliable. Broadcasts are non-atomic and might arrive at only a subset of the processes. We assume that the messaging delay is bounded: a message will arrive at its destination(s) within $MsgT$ after it was sent, or not at all.

Processes may fail at any point in time. We assume synchronous failure detection: if process p_i fails then all correct processes will be notified about this failure within $DetectT$, and the failure notification for p_i cannot overtake the last message that was sent by p_i. In case of multiple failures, the respective notifications may reach the correct processes in a different order. We also assume that once a process leaves the group, it behaves like a failed process or it explicitly rejects messages coming from the group. Finally, a process that fails or leaves the group may re-join the group at a later point in time, using a fresh unique process identifier.

Protocol state and messages

The state that is maintained locally by each process in order to run the protocol is summarized in Table 5.3, and the initialization procedure is given in Algorithm 1. The message

Table 5.3: Per-process protocol state.

state	Membership state ($JOIN, NORMAL, LEFT$)
myid	Identifier of the local process
coordid	Identifier of the coordinator process
grp	Group view including ticket information
seqno	Sequence number of last request sent/received
rmask	Reply bitmask for the current request
replies	Replies received from (ordinary) processes
reply	Reply of local (ordinary) process

Table 5.4: Protocol messages.

REQ[$pid, k, rmask, data$]**:** Request from pid (coord) with sequence number k for processes in $rmask$
RPL[$pid, k, data$]**:** Reply from pid for the request with sequence number k
JPOLL[pid]**:** Join poll from pid (coord)
JOIN[[pid]**:** Join request from pid, after a join poll
LREQ[$pid, rmask$]**:** Leave request from pid (coord) for processes in $rmask$
LACK[pid]**:** Leave request acknowledgment from pid
VPUSH[$pid, grpc$]**:** View-push request from pid (coord) for the new/updated group view $grpc$
VACK[pid]**:** View-push acknowledgment from pid

Algorithm 1 Protocol intialization

1: **function** INIT($id, pids$)
2:     $myid \leftarrow id$
3:     $seqno \leftarrow 0$
4:     $grp \leftarrow$ initGroupMembers&Tickets($pids$)
5:     **if** $pids = \varnothing$ **then**
6:         $state \leftarrow LEFT$
7:         $coordid \leftarrow 0$
8:     **else**
9:         $state \leftarrow NORMAL$
10:        $coordid \leftarrow$ minTicket(grp)
11:        **if** $myid = coordid$ **then**
12:            BECAMECOORD(procs(grp))
13:        **end if**
14:    **end if**
15: **end function**

---

types used for the different protocol components of GCBRR are shown in Table 5.4.

The membership status of the local process is kept in variable *state*, and can the the following values: $NORMAL$ if the process is a member of the group; $JOIN$ if the process is attempting to join the group (but is not yet a member of the group); $LEFT$ if the process has left the group (it is not a member of the group, and it is not trying to join the group).

The local group view, which contains the identifiers of the processes that are considered to be members of the group, is kept in the *grp* data structure. Importantly, *grp* also includes per-process ticket number information. The process with the smallest ticket number is the coordinator; its identifier is stored in *coordid*. At initialization, ticket numbers are assigned to the processes that are members of the group in increasing order of their identifiers. From that point onwards, the coordinator assigns the next ticket number to each new process that joins the group. The group view also contains information in order to differentiate between old and newly added members.

The sequence number of the last request sent/received (depending on whether the process is the coordinator or an ordinary process) is stored in *seqno*. The reply bitmask *rmask* encodes the (ordinary) processes addressed in the current request, which should send back a reply: if p_i should handle the request then the *i*th bit of *rmask* is 1 (else 0). The reply bitmap also defines the order in which these processes should reply: if the *i*th bit is the *k*th non-zero bit of *rmask* then p_i must send its reply in the *k*th transmission slot (after overhearing k replies). Ordinary processes store in *reply* their reply to the last request received; this may have to be re-transmitted, if requested by the coordinator. The *replies* structure is used by the coordinator to keep the replies that arrive from the ordinary processes.

Basic request/reply protocol

Algorithm 2 shows the basic request/reply protocol. When a new request-reply interaction starts, the sender/coordinator increases *seqno*, initializes *rmask* to address the target processes, broadcasts a request message, and waits for the replies to arrive or a timer to expire. The timeout is set as a function of $MsgT$, the number of processes that are expected to reply, and the application request handling delay. Each reply that is received is added to the *replies* structure, and *rmask* is adjusted accordingly. When the timer fires and some processes have not replied, the request is re-transmitted. This is repeated until all target processes either reply or fail. The coordinator proceeds with the next application request once the previous one has been handled to completion.

When a process receives a request, it checks *rmask* to determine whether it is among the addressed processes. If so, it compares the sequence number k with that of the last locally handled request *seqno*. If the request is new, it is handed over to the application, and the produce reply is stored in *reply*; else, the reply is already stored in *reply*. Replies are broadcasted, and as a result are received by other processes. The local process sends its reply following the order specified via *rmask*: if it is first in order, it does so as soon as the application produces the reply, else, right after it overhears the reply of the preceding

---

Algorithm 2 Basic request-reply protocol

 1: **function** REQUESTREPLY($dsts, data, procT$)
 2:     $replies \leftarrow \varnothing$
 3:     $seqno \leftarrow seqno + 1$
 4:     $rmask \leftarrow$ setBits($dsts \ominus myid$)
 5:     **while** $rmask \neq 0$ **do**
 6:         broadcast($REQ[myid, seqno, rmask, data]$)
 7:         $wt \leftarrow MsgT + procT+$ bitsSet($rmask$)$*MsgT$
 8:         **await**($rmask = 0, wt$)
 9:     **end while**
10:     **return** ($replies$)
11: **end function**

12: **function** OnRecv($REQ[pid, k, rmask, data]$)
13:     **if** isBitSet($rmask, myid$) **then**
14:         **if** $k \neq seqno$ **then**
15:             $seqno \leftarrow k$
16:             $reply \leftarrow$ PROCESSREQUEST($data$)
17:         **end if**
18:         $myturn \leftarrow$ false
19:         $wt \leftarrow$ posBit($rmask, myid$)$*MsgT$
20:         await($myturn =$ true$, wt$)
21:         broadcast($RPL[myid, seqno, reply]$)
22:     **end if**
23: **end function**

24: **function** OnRecv($RPL[pid, k, data]$)
25:     **if** $k = seqno$ **then**
26:         **if** $myid = coordid$ **then**
27:             $replies \leftarrow replies \oplus (pid, data)$
28:             $rmask \leftarrow$ clearBit($rmask, pid$)
29:         **else**
30:             $myturn \leftarrow$ nxtBit($rmask, pid, myid$)
31:         **end if**
32:     **end if**
33: **end function**

---

process. To deal with message loss, a timer is set to expire in order to send the reply at the corresponding transmission slot.

Join protocol

The join protocol is shown in Algorithm 3. The coordinator invites non-member processes to join the group, by broadcasting a join poll message. It then waits a certain amount of time for corresponding requests to arrive. Each time a response is received, the corresponding process is added to the local group view and it is assigned the next ticket number.

When the waiting time is over, the coordinator propagates the updated view to the group, via the view-push protocol (described in the sequel). In a first phase, the new view is disseminated to the old members (without the processes that are currently joining the group). In a second phase, the view is sent to each new member, in increasing ticket order.

A process wishing to join the group enters the *JOIN* state and waits for the coordinator to start a join poll. It then sends a join request with its identifier (this is a unicast message). The process remains in the *JOIN* state until it receives a group view from the coordinator. In the meantime, it might receive additional join poll requests, to which it responds as above. Processes that are already members of the group ignore join polls.

Leave protocol

The leave protocol is shown in Algorithm 4. The coordinator unilaterally decides which ordinary processes should leave the group and issues a leave request addressing only these processes. The leave request works along the lines of the basic protocol. The ordinary processes that are addressed by the leave request immediately enter the *LEFT* state, and stop handling (or explicitly reject) requests after sending back an acknowledgement. Once the leave request completes, the coordinator follows for each process that left the group the same procedure as for process failures.

View-push protocol

The goal of the view-push protocol is to commit a group view to a set of processes. It works along the lines of the request-reply protocol. In this particular case, the request carries the (updated) group view, and the target processes integrate it with their own local views, and send back an acknowledgment.

Fault handling & coordinator election

When a process is notified about the failure (or departure) of p, it removes p from its local group view. If p is an ordinary process, the coordinator stops waiting for replies from it. If p is the coordinator, the group member with the next smallest ticket number is elected as the new coordinator — without any further communication among the remaining group

---

Algorithm 3 Join protocol

1: **function** JOIN
2:     $state \leftarrow JOIN$
3:     **await**$(state = NORMAL)$
4: **end function**

5: **function** CHECKJOIN$(joinT)$
6:     broadcast$(JPOLL[myid])$
7:     $wt \leftarrow 2 * MsgT + joinT$
8:     **sleep**(tw)
9:     **if** new$(grp) \neq \varnothing$ **then**
10:         ViewPush(old(grp))
11:         **for each** $p \in$ sort(new(grp)) **do**
12:             ViewPush($\{p\}$)
13:         **end for**
14:     **end if**
15: **end function**

16: **function** OnRecv$(JPOLL[pid])$
17:     **if** $state = JOIN$ **then**
18:         unicast$(pid, JOIN[myid])$
19:     **end if**
20: **end function**

21: **function** OnRecv$(JOIN[pid])$
22:     $t \leftarrow$ nxtTicket(grp)
23:     $grp \leftarrow grp \oplus (pid, t)$
24:     GROUPUPDATED(grp)
25: **end function**

Algorithm 4 Leave protocol

 1: **function** FORCELEAVE(*pids*)
 2:     $rmask \leftarrow$ setBits($pids \ominus mypid$)
 3:     **while** $rmask \neq 0$ **do**
 4:         broadcast($LREQ[mypid, rmask]$)
 5:         $wt \leftarrow MsgT+$ bitsSet($rmask$)$*MsgT$
 6:         **await**($rmask = 0, wt$)
 7:     **end while**

 8:     **for each** $p \in pids$ **do**
 9:         OnProcessFailure(p)
10:     **end for**
11: **end function**

12: **function** OnRecv($LREQ[pid, rmask]$)
13:     **if** isBitSet($rmask, myid$) **then**
14:         $myturn \leftarrow$ false
15:         $wt \leftarrow$ posBit($rmask, myid$)$*MsgT$
16:         await($myturn =$ true, wt)
17:         broadcast($LACK[mypid]$)
18:         $state \leftarrow LEFT$
19:     **end if**
20: **end function**

21: **function** OnRecv($LACK[pid]$)
22:     **if** $myid = coordid$ **then**
23:         $rmask \leftarrow$ clearBit($rmask, pid$)
24:     **else**
25:         $myturn \leftarrow$ nxtBit($rmask, pid, myid$)
26:     **end if**
27: **end function**

Algorithm 5 View-push protocol

 1: **function** ViewPush(*pids*)
 2:      $rmask \leftarrow \text{setBits}(pids \ominus mypid)$
 3:      **while** $rmask \neq 0$ **do**
 4:          broadcast($VPUSH[mypid, rmask, grp]$)
 5:          $wt \leftarrow MsgT + \text{bitsSet}(rmask) * MsgT$
 6:          **await**($rmask = 0, wt$)
 7:      **end while**
 8: **end function**

 9: **function** OnRecv($VPUSH[pid, grpc]$)
10:      **if** isBitSet($rmask, myid$) **then**
11:          **if** $pid \neq coordid$ **then**
12:              $coordid \leftarrow pid$
13:              $seqno \leftarrow 0$
14:              $state \leftarrow NORMAL$
15:          **end if**
16:          **for each** $(p, t) \in grpc$ **do**
17:              $grp \leftarrow grp \oplus (p, t)$
18:          **end for**
19:          $myturn \leftarrow \text{false}$
20:          $wt \leftarrow \text{posBit}(rmask, myid) * MsgT$
21:          await($myturn = \text{true}, wt$)
22:          broadcast($VACK[mypid]$)
23:      **end if**
24: **end function**

25: **function** OnRecv($VACK[pid]$)
26:      **if** $myid = coordid$ **then**
27:          $rmask \leftarrow \text{clearBit}(rmask, pid)$
28:      **else**
29:          $myturn \leftarrow \text{nxtBit}(rmask, pid, myid)$
30:      **end if**
31: **end function**

Algorithm 6 Fault handling and coordinator election

1: **function** ONPROCESSFAILURE(pid)
2:     $grp \leftarrow grp \ominus (pid, *)$
3:     **if** $myid = coordid$ **then**
4:         $rmask \leftarrow$ clearBit($rmask, pid$)
5:         GROUPUPDATED(grp)
6:     **else if** $pid = coordid$ **then**
7:         $coordid \leftarrow$ minTicket(grp)
8:         **if** $myid = coordid$ **then**
9:             BECAMECOORD(grp)
10:             **if** new(grp) $\neq \varnothing$ **then**
11:                 ViewPush(old(grp))
12:                 **for each** $p \in$ sort(new(grp)) **do**
13:                     ViewPush($\{p\}$)
14:                 **end for**
15:             **end if**
16:         **end if**
17:     **end if**
18: **end function**

---

members. Before resuming normal operation, the new coordinator performs a view-push, if its view contains new members.

5.4 Request/reply semantics

The 1-N request/reply interaction supported by GCBRR is synchronous. This allows the coordinator to drive the group in a tightly-coupled way, enforcing progress at all (addressed) processes at the same pace while also keeping track of their aliveness.

From a fault-handling perspective, the request/reply is "exactly once" for processes that do not fail, and "at most once" for processes that fail. The coordinator is notified about the failures of ordinary processes via the *GroupChanged*() primitive. Alternatively, the coordinator can infer a process failure by inspecting the returned replies (recall that the *RequestReply*() does not return unless each of the addressed processes replies or fails).

Note that the request/reply is "non-atomic". This is because a newly elected coordinator does not attempt to complete the last request-reply interaction that was initiated by the old coordinator. Atomicity does not seem to make sense here, because it is unclear how the new coordinator could handle, in a meaningful way, the replies for a request that was not issued by it. Nevertheless, when an ordinary processes becomes the new coordinator, the higher layer is notified via the *BecameCoord*() primitive, and can perform any corrective actions as needed —in case of TeCoLa, a backup replica of the mission controller runtime evironment

is activated, becomes the new primary, and resumes application execution as discussed in Chapter 4.

5.5 Group management properties

The GCBRR protocol is designed for a group that has at least one correct process as a member at all times. If at some point all group members fail or leave, the group is destroyed and no longer exists. Also, a process cannot join a group that does not exist.

The protocol ensures the following properties related to the group management functionality:

- **Consistent ticket assignment (GM1)**: The assignment of ticket numbers to group members is the same at all processes that are members of the group.

- **Uniqueness of the coordinator (GM2)**: At any point in time, at most one and the same correct process can be the coordinator at all correct processes that are members of the group.

- **No election deadlock (GM3)**: As long as the group has at least one correct process as a member, a valid coordinator will be elected.

- **Join commitment (GM4)**: The confirmation to a joining process that it became a member of the group is binding.

Below we provide an informal argumentation for each property. We note that the leave protocol does not affect the essence of the group management functionality. Therefore, in the following we only consider process departures due to failures.

Consistent ticket assignment (GM1)

Let a group view consist of entries (p, t) where p is a process identifier and t the ticket assigned to that process. For any two group members p_1 and p_2 with local group views grp_1 and grp_2, it holds that: $(p, t_1) \in grp_1 \land (p, t_2) \in grp_2 \implies t_1 = t_2$. In other words, all members adopt the same ticket assignment for the processes that they consider to be members of the group.

This is easy to show based on how the protocol works. Assuming correct initialization, at the beginning, all the members of the group will have the same view and the same ticket assignment for all members. Once a process is assigned a ticket number, this never changes. When new members join the group, the coordinator assings different ticket numbers to them, and pushes the new view to all other group members which adopt the updated view (and new ticket numbers) without any modifications. As a result, even though during transition periods two processes may have different group views, if they both have the same process p in their views then p will have the same ticket number in both views.

Uniqueness of the coordinator (GM2)

From GM1, it follows that for any two correct (not failed) group members p_1 and p_2 it is guaranteed that $grp_1 = grp_2 \implies minTicket(grp_1) = minTicket(grp_2)$. Thus, GM2 trivially holds for any two correct processes that have the same local group view.

However, during the transition periods caused by group dynamics, some group members might have different views. More generally, let $(p, t) \in grp_1 \wedge (p, t) \notin grp_2$. We dicuss the different cases below.

Let us assume that the reason why p is in the view of p_1 but is not in the view of p_2, is because p failed and p_2 has detected the failure but p_1 has not yet detected this failure. If p does have the smallest ticket number in grp_1 and is not considered by p_1 to be the coordinator, this difference does not affect GM2. Else, if p_1 considers p to be the coordinator, again this does not affect the validity of GM2 as coordinator uniqueness only refers to the case where the coordinator is a correct process (that has not failed). Note that in any case p_1 will eventually detect the failure of p and will remove (p, t) from its view as well.

We now focus on views that differ only due to new processes that join the group. Starting from a state where all group members have the same local view, assume that one or more processes p join the group, and that the coordinator starts pushing the new view that includes the respective entries (p, t). Let the view push occur partially, reaching p_1 but not p_2, so that $(p, t) \in grp_1$ and $(p, t) \notin grp_2$. Since the view is first pushed to the old group members, p_1 cannot be a new member if p_2 is an old member. If both p_1 and p_2 are old members, due to the ticket assignment scheme it holds that $t_1, t_2 < t$, so any newly joined process p (even if this is included in grp_1 but is not included in grp_2) does not affect the choice of the coordinator at p_1 and p_2 which remains the same as before the join. Finally, if p_2 is a new member (one of the processes p that just joined the group), then the only way for $(p, t) \notin grp_2$ to hold is if p_2 has not yet received the view push from the coordinator. But in this case p_2 is not yet a member of the group (it is still in the JOIN state) and thus irrelevant for GM2 which concerns only processes that are members of the group.

No deadlock (GM3)

Given that GM2 holds, GM3 holds too, provided that if some correct process p_2 elects as the coordinator another correct process p_1 then p_1 will also consider itself as a member of the group. This is trivially so if p_1 is already a member when p_2 joins the group. It is somewhat less obvious if p_1 and p_2 join at the same time (respond to the same join poll). In this particular case, GM3 holds because the coordinator pushes the updated group view to each new member in increasing ticket order.

To see why, let us assume that the coordinator pushes the view grp_c that includes entries for the joining processes (p_1, t_1) and (p_1, t_2), with $t_1 < t_2$. Assume that the coordinator starts pushing the view in random order, but fails before completion. Also assume that all other old group members (which may have received grp_c) fail too. Finally, assume that p_2 has received grp_c and considers itself a valid group member, but p_1 has not received grp_c.

Then, p_2 will elect p_1 as the coordinator. However, p_1 remains in the $JOIN$ state, waiting for the next coordinator to initiate the next join poll, leading to a deadlock.

Join commitment (GM4)

In order for GM4 to hold, it must be shown that once a joining process p receives grp_c with an entry (p, t) for it, p can irreversibly enter the $NORMAL$ state and never has to fall-back to the $JOIN$ state. Equivalently, it must be shown that if p receives grp_c, every other process p_i with $(p_i, t_i) \in grp_c \wedge t_i < t$, which could take over as a coordinator while p is still alive, has already received grp_c. Indeed this holds due to the order in which the coordinator performs a view-push.

Also, p will not be addressed in the basic request-reply protocol, unless it considers itself as a member of the group (its state is $NORMAL$). This is guaranteed because the coordinator can proceed with the next request-reply interaction only after having successfully completed the corresponding view-push. It is also for this reason that a view-push is performed when a new coordinator takes over: to ensure that new members are aware of their membership status. In principle, a view push is not strictly required (and could be performed in a lazy fashion) in case processes fail or leave the group, and the new group view does not include any new members.

5.6 Key performance properties

The critical path of GCBRR in terms of performance is the basic request-reply protocol. This is designed to achieve several important properties:

(1) It is contention-free, avoiding concurrent transmissions from different processes.

(2) It incurs the minimum number of transmissions needed for a 1-to-N request-reply interaction: the request and each reply are transmitted just once (assuming no loss).

(3) It achieves minimal latency since ordinary processes send their replies as fast as possible, according to the schedule specified by the coordinator.

(4) It is offers very predictable performance, and allows the system to operate close to the channel capacity.

(5) It enables robust message recovery in case of loss, by addressing only the processes that have not responded.

(6) The values for the message transmission delay and application-level request processing delay can be chosen generously, and affect performance only in case of message loss.

(7) Last but not least, no assumptions are made about the underlying medium-access-control (MAC) mechanism. Therefore GCBRR can work very well even on top of simple radios that do not have an intelligent MAC.

Note that, in principle, the join protocol is subject to contention. This can happen if a large number of processes wish to join the group at the same time. We consider this to be an exception rather than the rule. Even then, a simple back-off mechanism can greatly reduce the probability of collisions. But it is important to stress that the join protocol does not interfere with the basic request/reply protocol, because processes that wish to join remain fully silent until they receive a join poll. It is up to the coordinator to activate it when desired, via the respective primitives.

5.7   Evaluation

We have implemented GCBRR for a Linux-based platform, as a user-space library that resides on top of the operating system. Our implementation uses RAW sockets (bypassing the IP stack). This section presents a performance evaluation over WiFi, and discusses the most important results.

Experimental setup

We test our prototype implementation using two different setups, a simulated 802.11 network and a 802.11 testbed. Note that this evaluation was conducted before the design of the TeCoLa programming model, so the protocol was developed as a proof-of-concept without implementation optimizations such as zero-copy message passing and efficient lock-free techniques as in the version that actually supports the TeCoLa software stack. For this reason the absolute numbers regarding the critical path of GCBRR may appear larger compared to the ones in TeCoLa however this does not affect the validity and the focus of this evaluation. We performed a number of indicative experiments with the newer version of GCBRR to confirm that the trends and the results of the comparisons bellow are the same with the previous version and we conclude that the repetition of the entire experiment set would not produce any additional value to this evaluation.

Simulated setup: For the simulated setup we use the OpenNet simulator [37]. This is an open-source simulation environment, which is build on top of Mininet [74] and ns-3, combining their features to provide a realistic virtual wireless testbed infrastructure on a single PC. OpenNet uses Mininet to create a network of virtual Linux hosts, which can run conventional applications without any modification. Each virtual host features a virtual network interface which is internally connected via a TAP device to an ns-3 node. In turn, ns-3 provides the modeling of the 802.11 wireless channel between virtual hosts. To reflect the testbed setup, we configure ns-3 for the 802.11g protocol with a transmission rate of 1 Mbps for both unicasts and broadcasts.

Testbed setup: For the real measurements we use the NITOS testbed [3] featuring ICARUS [5] wireless nodes. These are PC-class machines with an Intel Core i7-2600 Processor with 4 GB RAM and wireless Atheros 802.11a/b/g/n cards, running Linux. The testbed environment is protected from external traffic/interference. The nodes are placed in a grid, and can communicate with each other in 1 hop. The WiFi interface is accessed through the standard socket interface via *ath9k* network driver, and is configured to operate with the 802.11g protocol in ad-hoc mode. To make a fair comparison among broadcast-based and unicast-based approaches, we fix the transmission rate to 1 Mbps for both. It would also be possible to transmit broadcast packets at the highest rate of the wireless interface by using the pseudo-broadcast transmission approach presented in [65], but this would not contribute to the essence of our evaluation (the main point is for broadcasts and unicasts to be equally fast). The average round-trip time between two nodes is about 26 milliseconds for packets with a payload of 1500 bytes.

The simulated setup is used to perform experiments across a wide range of parameters, without requiring physical nodes. We then use the testbed for testing selected scenarios in order to verify the trends we observed via simulations. In both cases, each process (group member) is placed on a different virtual/physical node.

Basic 1-N request-reply

To evaluate the raw performance of the basic request-reply interaction of GCBRR, we use an application that issues dummy requests and produces dummy replies, without performing any processing. Request and reply messages are filled with dummy data so that all transmitted packets have the maximum payload size (1500 bytes).

We compare GCBRR with four reliable point-to-point transports: TCP-SEQ, TCP-PAR, RUP-SEQ and RUP-PAR (RUP stands for reliable unicast protocol). In the TCP variants, the coordinator keeps a separate connection with each ordinary process, which is reused as a transport channel for all 1-N reuest/reply interactions. RUP variants use plain datagrams over RAW sockets, with a simple acknowledgment scheme for the re-transmission of requests. SEQ variants perform the request-reply interaction sequentially, addressing one ordinary process at a time; this avoids contention but comes at increased latency. In the PAR variants, the coordinator first sends the request to all ordinary processes, and then waits for their replies; this can reduce the total latency but introduces some contention.

We perform experiments for groups of different sizes: 3, 6 and 12 processes (coordinator and ordinary processes). To stress the network, we let the coordinator issue 1000 request-reply interactions at full speed. We record the total throughput and latency of each interaction, measured at the coordinator. The results are shown in Figure 5.4 and Figure 5.5, respectively.

As can be seen, GCBRR performs significantly better than all other variants, and the difference increases with the group size. For 3 processes, GCBRR achieves $1.62x$ and $1.35x$ the throughput of TCP and RUP variants, going up to $2.22x$ and respectively $1.76x$ for 12 processes. This is because GCBRR transmits fewer packets and scales better for a larger
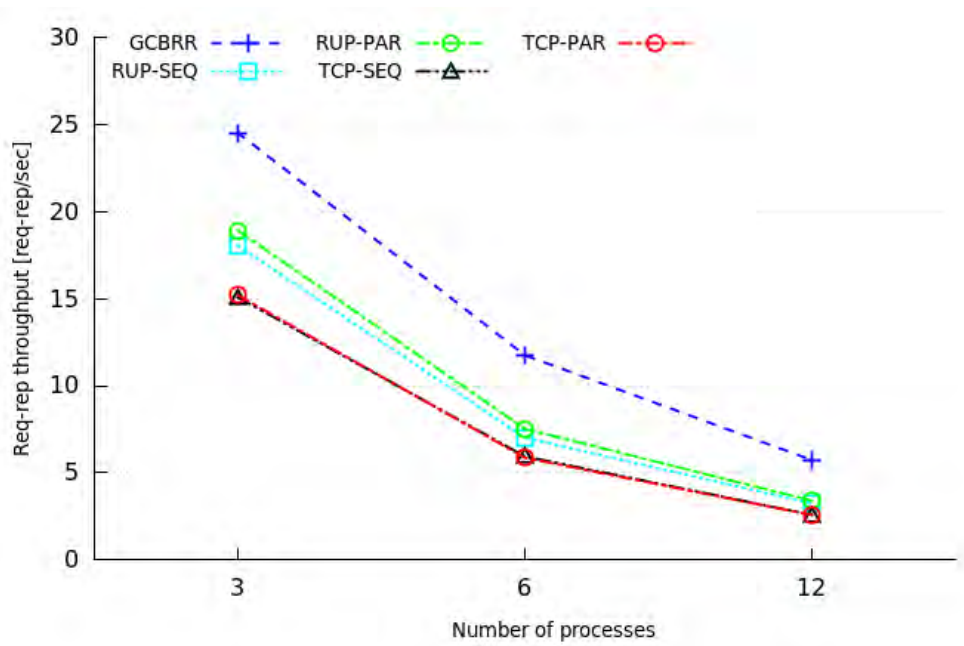
Figure 5.4: 1-to-N request-reply throughput for different group sizes.
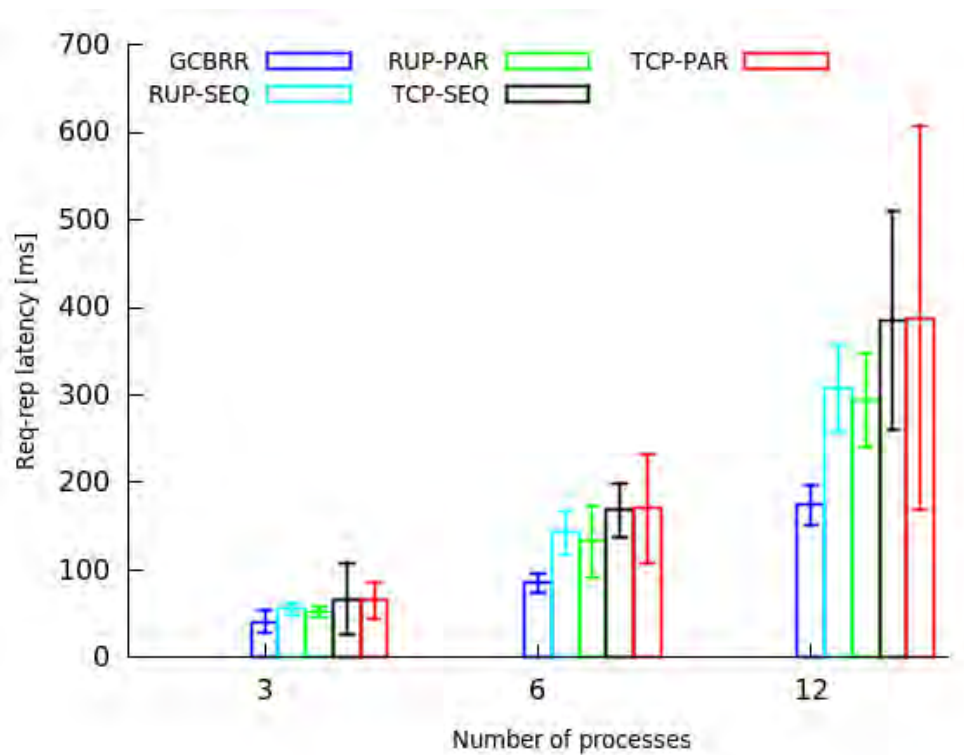

Figure 5.5: 1-to-N request-reply latency for different group sizes.

number of processes. Note that for each unicast WiFi sends a MAC-level acknowledgement, which further increases the overhead of point-to-point variants. TCP achieves lower throughput than RUP due to the extra (TCP-level) acknowledgments, yet this difference becomes less pronounced for larger group sizes. Message loss is negligible in all cases, as the WiFi flow-control mechanism kicks in when the network is stressed.

The difference in the number of packet transmissions also reflects on the latency of the request-reply interaction. Again, GCBRR outperforms all other variants, especially in larger groups. For 12 processes, its latency is only about 1/2 and 2/3 that of TCP and RUP, respectively. RUP variants are generally better than TCP. Note that RUP-PAR has a slightly lower latency than RUP-SEQ because of the concurrent transmission of replies back to the coordinator. This advantage does not show in the TCP variants, due to the higher protocol overhead of TCP. In fact, TCP-PAR has a much larger variance than TCP-SEQ. This is an effect of the increased contention: the message transmissions are slowed-down due the lower-level signalling and flow control mechanisms of WiFi.

One-way N-1 message flow

We also evaluate GCBRR regarding its ability to support a reverse, one-way message flow, from ordinary processes to the coordinator. In this case, the coordinator polls ordinary processes via an empty request, and receives their messages as replies. For comparison, we use TCP-PAR and RUP-PAR, where each ordinary process sends its messages to the coordinator over a TCP connection and a RAW socket, respectively. In RUP-PAR, messages are acknowledged using an alternating bit scheme. As an additional reference, we use the simplest possible best-effort approach where messages are sent via unreliable unicasts over a RAW socket, referred to as unreliable unicast protocol (UUP-PAR).

We measure the maximum message throughput that can be achieved in a group of 3, 6 and 12 processes, at full speed. In GCBRR, the coordinator polls the ordinary processes 1000 times, while in TCP-PAR, RUP-PAR and UUP-PAR we let each process send 1000 messages to the coordinator. The aggregate throughput is measured at the coordinator, by recording the time between the arrival of the first and the last message. At each ordinary process, we record the time that elapses between two consecutive message transmissions, which essentially reflects the messaging latency. Figure 5.6 shows the results obtained for the aggregated throughput. Figure 5.7 shows the latency recorded at each process in an indicative run with a group size of 12 processes.

Despite the polling overhead, GCBRR achieves higher message throughput than TCP-PAR and RUP-PAR, even for smaller groups. The difference increases for larger groups, where TCP-PAR and RUP-PAR lead to increased contention, and the relative cost of polling decreases. At 12 processes, the throughput of GCBRR is $1.28x$ and $1.13x$ that of TCP-PAR and RUP-PAR, respectively. In this case, GCBRR even outperforms UUP-PAR which even has a very significant message loss of about 7%. Furthermore, unlike the RUP and TCP variants, GCBRR has a stable and predictable messaging delay. In fact, with TCP-PAR all nodes occasionally experience huge delays. Due to contention, TCP-PAR has some message
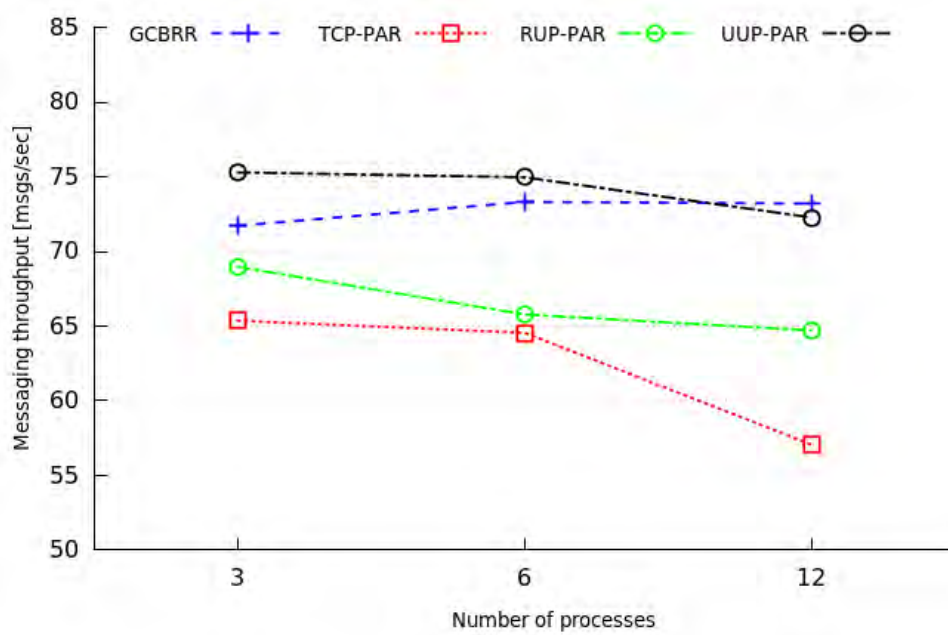
Figure 5.6: Aggregate N-to-1 message throughput for different group sizes.
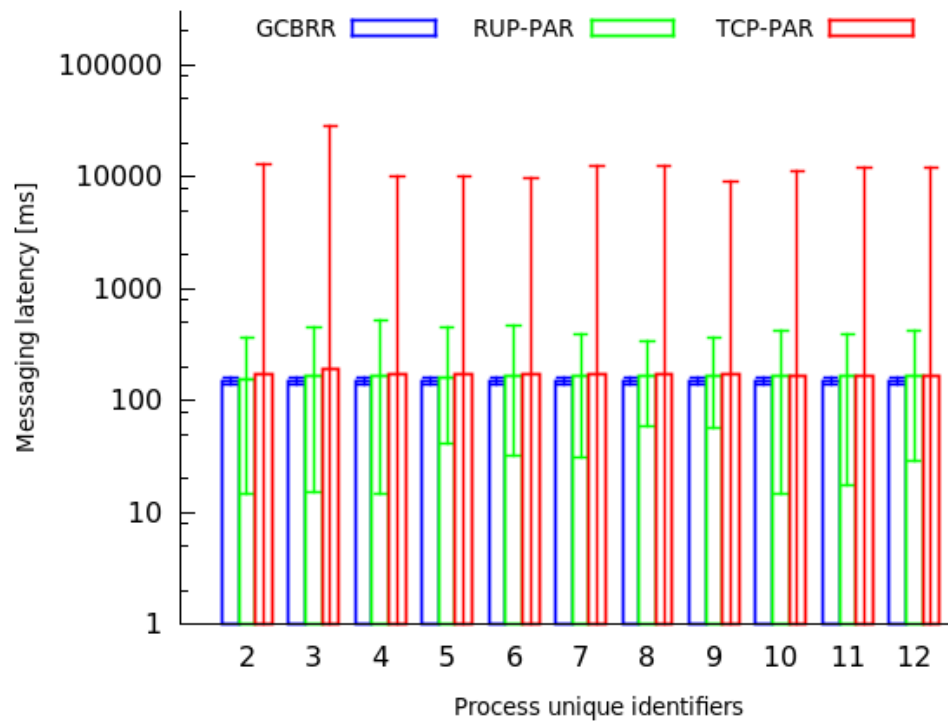

Figure 5.7: Per-node messaging delay for an indicative run with 12 processes. Note that the scale of the y-axis is logarithmic.
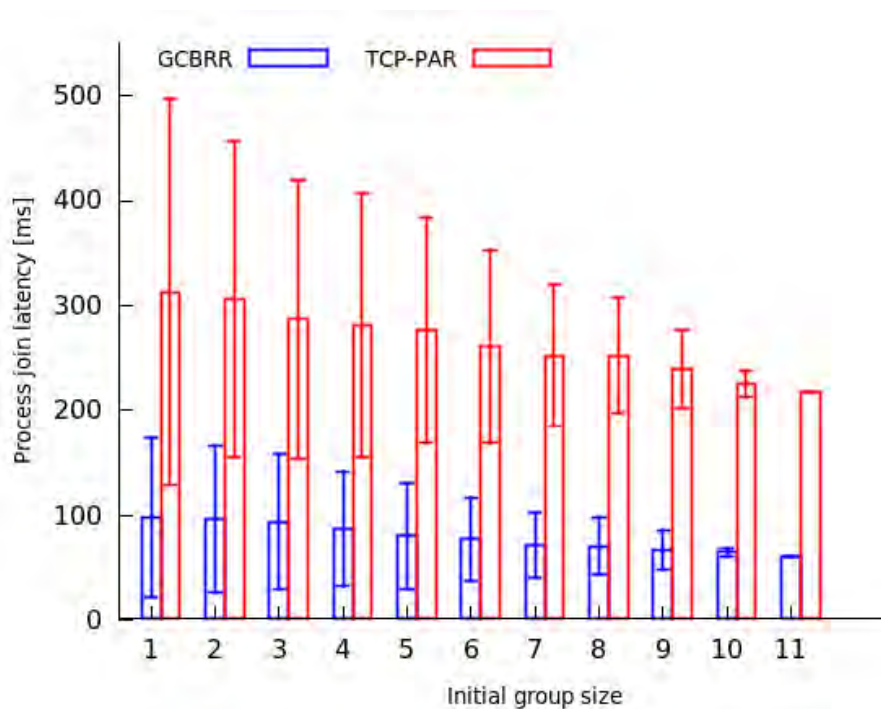
Figure 5.8: Node join delay when starting from initial group size N and adding $12 - N$ processes in one shot.

loss, which leads to retransmissions and the activation of the TCP flow-control mechanism. In contrast, GCBRR and RUP-PAR did not need to perform any retransmissions. Also note that in TCP-PAR message sending sometimes appears to be instantaneous due to internal buffering.

We wish to note that, in general, polling is not ideal for supporting sporadic messaging from multiple sources (ordinary processes) to a destination (coordinator). In this case, it is better to use a conventional uncoordinated approach. However, polling via GCBRR can be a valid option if the rate at which ordinary processes wish to send messages to the coordinator is not very low. In Chapter 6, we investigate this problem a bit further, and propose an approach that can be used to infer the application messaging rate and dynamically adjust the polling rate of the coordinator accordingly.

Group management

Finally, we evaluate GCBRR in terms of the group management overhead. As a reference, we use a TCP-based approach where the coordinator keeps an open connection with each ordinary process. A joining process opens a TCP connection to the coordinator when it receives a join poll (without sending a join request). It considers itself as a member once it successfully receives the group view over the TCP connection. The coordinator asks a process
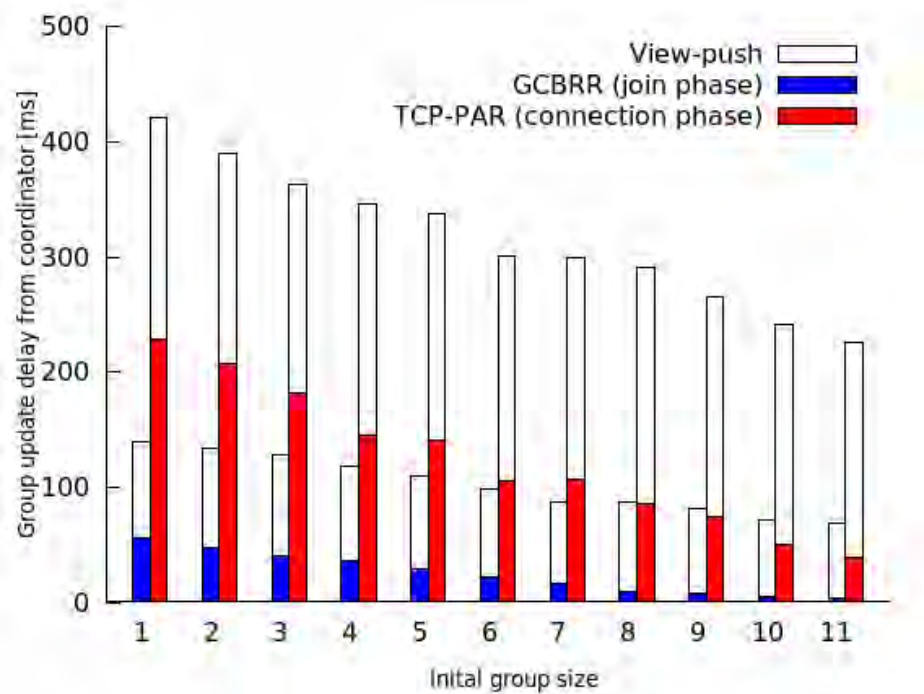
Figure 5.9: Group update delay, when starting from initial group size N and adding $12 - N$ processes in one shot.

to leave by sending a request over the corresponding TCP connection. A process failure is detected when the TCP connections breaks. Like in GCBRR, the coordinator performs a view-push but in this case the group view is sent to each member over the respective TCP connection. When an ordinry process detects the failure of the coordinator, it elects the new coordinator and opens a connection to it; if it elects itself, it waits for all other ordinary processes to connect to it. As an implementation detail, we note that the coordinator uses a single thread for accepting new connections via a network event loop based on the *poll* Linux system call, similar to the technique described in [38].

We conduct a series of experiments to chart the cost of the join operation as a function of the group size and number joining processes. In each experiment, the group starts with a different initial size, and then more processes join so that the group ends up with a total of 12 processes. The new processes join the group at once, via a single join poll. At each joining process, we record the join latency as the time that elapses between the reception of the poll message and the reception of the group view from the coordinator. At the coordinator, we measure the group update delay, which is the time it takes to receive all join/connection requests and push the updated view to all ordinary group members. Figure 5.8 shows the results for the join latency, Figure 5.9 reports the group update delay.

With both approaches the average join latency and deviation naturally decreases as fewer processes join the group. However, the join latency and group update delay for GCBRR is

only 1/3 that of the TCP-based approach. This is because the TCP variant has an extra connection setup overhead, which becomes significant when the number of joining processes becomes large. Also, it sends the group view to each member separately, whereas GCBRR does this with a single transmission for old group members, using the basic 1-N request/reply support.

We have not performed any experiments for the leave operation. In GCBRR this is very fast. It it boils down to a single 1-N request/reply interaction whose performance follows the same trend as shown in the previous experiments (actually, it is faster because in this case the request and reply messages are practically empty). Also, the cost for a coordinator switch/election, once its failure has been detected, is negligible: it is the time it takes for a process to pick the smallest ticket number from its local view.

5.8 Related work

The reliability of multicast communication over a shared medium have been studied mainly in the lower level of the network stack.

In [73] authors propose two multicast protocols which are relying on the IEEE 802.11 CSMA/DA and RTS-CTS mechanism in order to increase the reliability of broadcast transmissions. Both protocols support a single sender and assume a microcell-based wireless network where each cell is managed from a base station located in the center of the cell. The first protocol (LBP), is a leader based protocol while the other (PBP) is a probabilistic feedback-based protocol. In LBP, a sender claims the channel and sends a broadcast message to the receivers in the group. Among the members of the reception group, an elected leader sends a single positive acknowledgment to the sender. The rest of the receivers only send a negative acknowledgment in case of erroneous reception; this is sent immediately to destroy the positiveacknowledgment of the leader. The sender re-transmits a message if it does not receive a positive acknowledgment within a specified period. PBP employs a CTS/RTS scheme where a sender transmits a multicast-RTS and waits to receive CTSs from the participant receivers in order to transmit the data. In order to avoid collisions, the receivers back-off with a certain probability which is based on the number of group members before the CTS transmission. Results show that both protocols increase broadcast reliability while LBP achieves higher throughput compared to PBP. However both protocols do not guarantee collision avoidance as in our case and require strict clock synchronization.

A similar approach is followed in LM-ARF [43]. This assumes an elected leader which is responsible to acknowledge mulitcast transmissions like in LBP, and makes use of CTS/RTS frames in order to increase the fairness between unicast and broadcast flows. Moreover, LM-ARF performs adaptation of the transmission rate based on the ARF [64] scheme of 802.11 to avoid inefficient broadcast transmissions when network saturates to its capacity. LM-ARF protocol inherits the time synchronization requirements from LBP and is even more closely coupled to the underlying MAC protocol. In contrast, our protocol is MAC-neutral and

takes advantage of the request-reply communication pattern without involving additional control frames.

The Broadcast Medium Window method (BMW) [98] provides a reliable broadcast solution. Briefly, BMW maintains three lists, the neighbor list, the send buffer and the receiver buffer. The neighbor list contains the addresses of the neighbor nodes. The send buffer holds the transmitted multicast frames for potential retransmissions, and the receiver buffer has the sequence numbers of the received messages. In BMW, the sender performs a collision avoidance phase and sends an RTS which contains the current and the last sequence message number to one of its neighbors. The receiver inspects the sequence numbers of the RTS and requests any missing messages via its CTS reply. When the sender receives the CTS, it sends the data and waits for an acknowledgment. Other nodes that receive the CTS yield until the reception of the data acknowledgement. This continues for all neighbor nodes. Note that when network saturates to its capacity and severe loss occurs the protocol reverts to the usual unreliable broadcast transmissions mode of 802.11. BMW provides a reliable solution for broadcast, but requires a large number of messages and creates contention phases equal to the number of neighbors.

Complementary to the BMW method is the Batch Mode Multicast MAC (BMMM) [94]. BMMM also uses of the RTS/CTS mechanism but eliminates the multiple contention phases of BMW to a single one. More specifically, BMMM introduces a new frame type called RAK (Request for ACK) which is similar to ordinary MAC-level ACKs. The sender of a multicast competes for the channel only at the start of the multicast transmission with its RTS. The RAK frame follows directly after the RTS, instructing the receivers to send their CTS and ACKs in order. BMMM follows an approach similar to our protocol with respect to the coordinated return of the replies, but involves more messages and abandons reliability when the network is stressed. Also, in case of a message loss, the sender competes for the channel again.

Another approach that improves broadcast reliability is BSMA approach [97]. BSMA assumes that the radio hardware integrates DS (direct sequence) capture ability in order to lock-on a strong signal in the presence of interference. To this end, a collided packet with stronger signal strength will be correctly received while others will be silently dropped. The protocol utilizes the 802.11 RTS/CTS frames and negative acknowledgments. Briefly, the sender executes a collision avoidance phase, broadcasts its RTS frame and waits to receive CTSs from the group. The receivers respond to RTS with their CTSs then wait to receive the data. Once the sender receives a CTS, it transmits the data. During the wait phase none of the receivers is allowed to compete for the channel. In case of message loss receivers reply with NAK prompting a re-transmission. The results show that throughput and reliability increases over the traditional unreliable broadcast but the protocol relies strictly on 802.11 collision avoidance and requires specific radio capabilities. Our protocol is MAC-neutral while it is not based on any specific radio functionality.

An approach independent of the underlying MAC is presented in [34]. The proposed scheme is based on the reliability that is provided from the upper layers of the network stack and particularly from IP. The scheme assumes an infrastructure-based wireless network

consisting of an access point (AP), at least one assistive station (AS) and other ordinary stations. Each station is equipped with a passive dongle that monitors the wireless traffic by capturing the exchanged network frames. The AS has a well known IP address and each multicast transmission is performed from the AP towards the AS through plain IP datagrams. Other participants receive the multicast message by overhearing these transmissions.

The use of physical broadcast in order to increase throughput has been also studied in mesh network architectures. COPE [65] makes use of physical broadcast on par with network coding for effective network utilization. Besides the coding techniques, COPE takes advantage of the wireless broadcast nature by employing a low level pseudo-broadcast. Specifically, a sender broadcasts a packet by sending a unicast to a particular node. The broadcast is received by other participants through the unicast transmission overhearing. The increased reliability of unicasts provides increased reliability for broadcast as well.

The works above improve the physical broadcast reliability, however they do not have predicable performance which is a useful property for many latency sensitive applications. They also target general one-way multicast communication where the receiver merely sends a low-level acknowledgement, as opposed to GCBRR which works in an end-to-end fashion [91] and where replies may carry application-level information. As shown in previous Chapters, GCBRR can serve as a foundation for higher-level schemes that require reliable and fast 1-N request/reply communication, such as primary-backup replication [35], remote procedure calls [75], and team programming over a wireless network as in TeCoLa or [79].

Chapter 6

Adaptive polling method for N-1 data flows

As discussed in Chapter 5, besides the 1-N request/reply interaction, GCBRR can also be used as a transport for an information flow in the reverse N-1 direction, from the ordinary processes to the coordinator, via polling. This chapter presents a method for the dynamic adaptation of the polling rate based on an application-agnostic estimation of the process data rates. We begin by motivating the approach in a more general context. Then we discuss the proposed approach and how this is implemented on top of GCBRR. Finaly, we provide a performance evaluation of our method for indicative system configurations using real wireless sesor nodes and we dicuss the related work.

6.1   Motivation

GCBRR is designed primarily to support a coordinated 1-N request/reply interaction pattern in an efficient way that does not rely on MAC-level mechanisms to avoid, detect and deal with contention among the communicating processes. However, it can also be used to support a reverse N-1 information flow, again with zero contention. While TeCoLa does not implement such purely unidirectional data flows from the nodes that are part of the mission group to the mission controller, this usage scenario GCBRR can be attractive in a different system context, in particular when data has to be collected from multiple sensor nodes over a low-bandwidth wireless channel.

Indeed, a wide range of modern monitoring applications, spanning from personal activity tracking to smart homes, employ wireless sensor nodes to obtain the required measurements. In most cases, the sensors are sampled at a given rate, and the data, possibly after some local processing, is sent over the air to a collector, which, in turn, stores and processes this data to support some monitoring or decision/control process. Several applications require this data transmission to be reliable. Also, in some occasions, data may need to be sent to the collector at a relatively high rate, close to the capacity of the (typically low-bandwidth)

wireless channel.

There are two fundamentally different ways to achieve this. One approach is for each sensor node to send data to the collector as soon as this is produced. When the collector receives the data, it replies with an acknowledgement. We refer to this as *notification*. The other approach is for the collector to explicitly request the sensor nodes to send their data. In this case, each node stores the data in a local buffer, and transmits it only at the request of the collector. If the node receives a request but has no data, it replies with an empty acknowledgement. We refer to this method as *polling*.

Notification is attractive when there are a few sensor nodes and the data generation rate is relatively low compared to the capacity of the wireless communication channel. Else, it can result in bad performance and poor effective channel utilization. This is particularly so when nodes transmit in an uncoordinated way and the wireless communication technology does not feature any advanced medium access control mechanism.

In this case, polling can provide very good performance, especially when the data rate approaches the channel capacity, without requiring nodes to feature advanced radios or to be synchronized with each other. However, a key issue with polling is that the data generation rate may be unknown, and as a result the coordinator may poll too frequently or too slowly. Both situations are undesirable as they can waste system resources or reduce data freshness and lead to data buffer overflows, respectively. To address this issue, we propose a method for adapting the rate at which the collector polls the sensor nodes. As a basic transport service, we use GCBRR, with minor extensions that are discussed in the sequel.

6.2 Basic polling protocol

We formulate the polling protocol along the lines of GCBRR, where the coordinator process takes the role of the data *collector* and the ordinary processes are the *sensors* that produce the data of interest.

Let processes $p_i, 1 \leq i \leq N$, represent the sensors, and let a special process p_0 be the collector. All processes are in the same broadcast domain, with maximum packet transmission delay $MsgT$. When the collector p_0 wishes to retrieve data from the sensors, it starts a poll by broadcasting a poll request. The payload of the poll request consists merely of a bitmask $rmask[1..N]$ that plays the same role as in the GCBRR protocol. More specifically, if $rmask[i] = 1$ then p_i is expected to reply with a data packet, else if $rmask[i] = 0$ then p_i should remain silent during the poll. This bitmask also indicates the order in which sensors should respond: if both p_i and p_j are included in the poll ($rmask[i] = 1$ and $rmask[j] = 1$) and $i < j$, then p_j must wait for p_i to send data, before it performs its own transmission. Figure 6.1 illustrates the message exchanges that take place between the collector and sensors during a poll.

Every sensor process keeps the data it generates in a local buffer, and sends it when polled (several data items can be piggybacked in the same packet). The data packet also carries information about additional data items (beyond the ones included in the reply) that
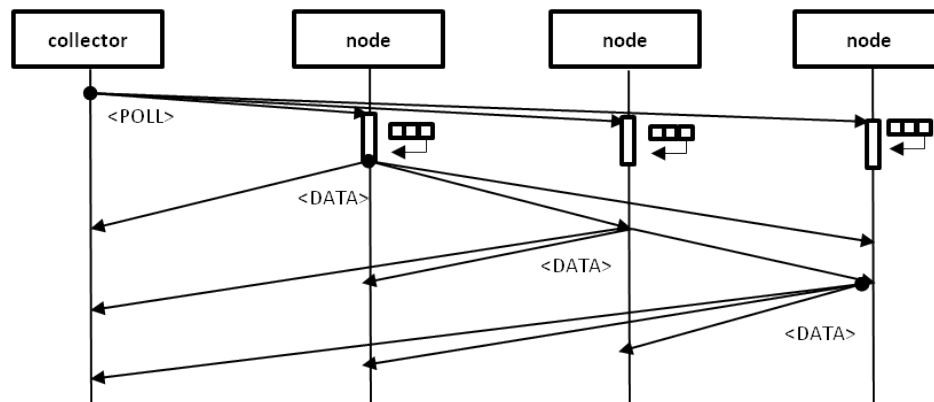
Figure 6.1: Basic polling protocol. When polled, nodes send data in sequence, in ascending order of their identifiers.

are in the sensor's buffer. If a sensor is polled and its buffer is empty, it replies with an empty data packet.

Note that the collector may need to perform several consecutive polls in order to retrieve all the data that is available in the buffers of the sensors. These polls are performed at full speed, adjusting the address mask as needed, until the collector receives a reply from every sensor and all sensors indicate that their data buffer is empty. We refer to such a batch of consecutive polls as a *poll cycle*.

To get a feeling about the performance of this approach, Figure 6.2 shows the maximum sustainable aggregate and per-node reliable data transfer rate over IEE 802.15.4 radio, as a function of the number of sensor nodes that generate/send data. As a reference we use an optimal transport protocol where the sensor nodes send their data to the collector with prefect medium access synchronization. We also show the maximum data rate that can be sustained with acceptable data loss (less than 1%) for a notification protocol where each sensor node unicasts its data to the collector and waits for an acknowledgement.

It can be seen that the polling protocol approaches optimal performance as the number of sensor nodes increases, because the overhead of the poll request is amortized over a larger number of data packet transmissions. In contrast, the performance of the notification protocol is bad and degrades to the number of sensor nodes, due to the increasing contention and number of collisions.

6.3 Rate Estimation

If the collector knows the rates at which sensors generate data, it can adjust its polling strategy accordingly. However, in the general case, the data rate r_i of p_i may be unknown, even to the sensor process itself. For instance, the application may not inform the protocol layer about the rate at which data will be generated, because it may be not be able or not
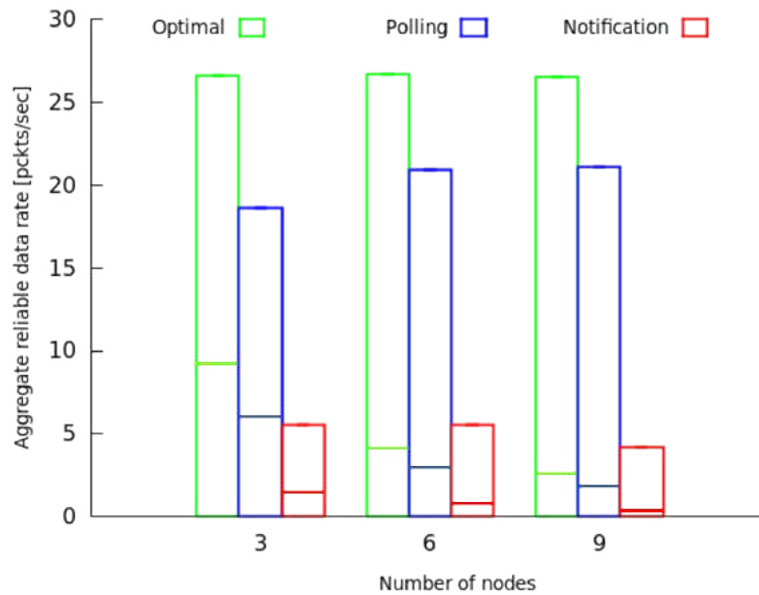
Figure 6.2: Maximum aggregate (full bar) and per-node (sub-bar) rate for reliable data transfer.

willing to compute it. The challenge is for the protocol layer to estimate r_i without any input from the application.

We assume that data generation is independent for each sensor, and let each p_i estimate r_i locally. This estimate is then piggy-backed to the response p_i sends to the collector when it is polled.

In the general case, the average data rate r_i may vary in time. We assume that changes in the average data rate are rather abrupt, corresponding to internal state changes of the application (which are hidden from the protocol layer). We also assume that once such a state change takes place, the application remains in that state for a relatively long time. These assumptions are realistic for a wide range of applications. Figure 6.3 illustrates two indicative scenarios for the data generation models we consider in our work. The top part shows a process that performs a transition between different but constant data rates, while the bottom part illustrates a transition between two different average rates for a stochastic data generation process.

Our goal is to design a unified method that can estimate the average rate in both cases, without prior knowing which case actually applies to the current system operation. To describe our approach we introduce additional notations that capture the time-dependency of the average data rate. We denote with $r_i[n]$ the actual average data rate of the application process when the n-th data packet is generated.

Process p_i can estimate $r_i[n]$ in different ways depending on the assumptions about the model that governs traffic arrivals. For our purposes, we assume a first-order dynamical
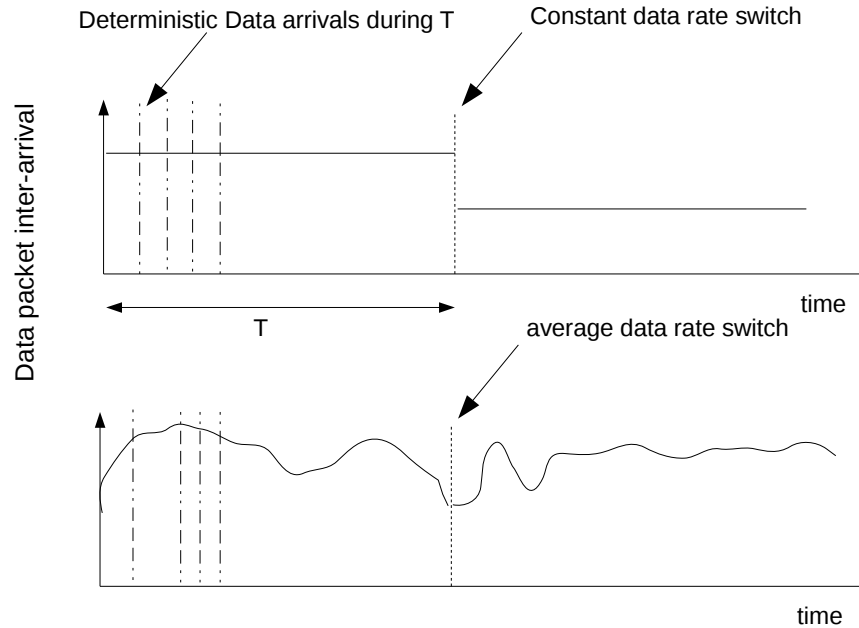
Figure 6.3: Transitions between different data rates, for periodic (top) and stochastic data generation (bottom).

model for the the evolution of the average data rate:

$$r_i[n] = a \times r_i[n-1] + u[n], \ n \geq 0 \qquad (6.1)$$

The value of parameter a corresponds to the weight of the last average rate of the application: if a is close to 0 then the previous value hardly affects the next value, whereas if a has a value close to 1 then the average rate changes slowly over time. In this model $u[n]$ is a random variable that follows a Gaussian distribution, and models potentially abrupt changes in the average rate over successive time instants. Since our assumption is that most of the time the application generates data at a stable average rate, the best choice is to set a close to 1.

We want to compute an estimate of $r_i[n]$. However, this estimate will unavoidably suffer from measurement noise: $x_i[k] = r_i[k] + w[k]$. The question is how to estimate $r_i[n]$ accurately. The metric we use is the mean square error (MSE) from the Bayesian estimation theory.

Since $u[k]$ is Gaussian for the linear model in (6.1), we adopt the scalar Kalman filter, which has been shown to be optimal for the linear data model [66]. The scalar Kalman

algorithm is an adaptive filter defined from a set of recursive equations:

$$\text{Prediction}: \hat{r}[n|n-1] = a\hat{r}[n-1|n-1]$$
$$\text{Min. Pred. MSE}: c_e[n|n-1] = a^2 c_e[n-1|n-1] + \sigma_u^2$$
$$\text{Kalman Gain}: K[n] = \frac{c_e[n|n-1]}{c_e[n|n-1] + \sigma_u^2}$$
$$\text{Correction}: \hat{r}[k] = \hat{r}[n|n-1] + K[n](x[n] - \hat{r}[n|n-1])$$
$$\text{Minimum MSE}: c_e[n|n] = (1 - K[n])c_e[n|n-1]$$

The algorithm initializes $c_e[n-1|n-1]$ to an arbitrarily low value close to 0. Then it calculates recursively $K[n], \hat{r}[n], c_e[n|n]$.

The above model will work well, regardless of whether data generation is periodic or stochastic (Poisson), provided that the average data rate remains constant, and a is set close to 1 (which is true in our case). However, as already noted, we are interested in supporting scenarios where abrupt changes in the average data rate can occur. One way to do this is to employ a more intelligent filter that can detect and respond to such transitions. Here, we follow a different approach, namely we keep the same simple filter, but *control* its operation based on information that is available at the layer of the polling protocol. The heuristic is rather intuitive: since the above filter is not designed to track significant changes in the average data rate, it is reset whenever there is an indication of inefficient polling. More concretely, each sensor process resets its local Kalman filter (i) when the collector performs several consecutive (void) poll cycles that do not return any data, or (ii) in a given poll cycle the collector performs several consecutive polls in order to empty the buffers of the data processes. As will be shown in the next section, it suffices to adapt the operation of the filter, and to let the collector adjust the polling rate according to the data rate estimates received from the sensor nodes.

6.4 Experimental Evaluation

We have evaluated the proposed polling method using real wireless nodes that communicate over 802.15.4 radios. In the following, we describe the test setup, and then present results from indicative experiments.

Experimental setup and metrics

For our experiments we use TelosB-class devices from AdvanticSys [6], equipped with 802.15.4 radios. To enable detailed logging during the experiments, the protocol logic runs on powerful PCs, which are connected over serial to the TelosB devices that essentially act as wireless modems. We have confirmed that the serial interface is fast enough to support the packet rate of 802.15.4. We refer to a PC-TelosB pair as a node.

The nodes are arranged in a 1-hop topology so that they can communicate with each other directly. A distinguished node is the collector, all others play the role of sensor nodes that produce/send data. Sensor nodes run a data generator application, which produces data items according to a predefined scenario. The scenario is stored in a file as a sequence of time intervals. When the experiment starts, the application waits for a short random back-off interval and then, in a loop, reads the next entry from the scenario file, sleeps for the specified amount of time, and produces the next data item that should be eventually sent to the collector. We investigate both periodic/deterministic and stochastic data generation scenarios. In the latter case, the scenario file is generated automatically using Matlab, according to the behavior of a Poisson process (separate scenario files are genenerated for each sensor node). The data produced by the application are passed to the protocol layer for transmission. In our experiments we let each data item occupy a full packet. As a consequence, it is not possible to piggy-back several data items in a single packet, and if a sensor node has more than one items buffered locally then the collector has to retrieve them via a corresponding number of consecutive polls.

The protocol layer of the sensor nodes runs the heuristic for estimating the local application data rate, as discussed in Section 6.3. This information is included in the data packets sent to the collector when the sensor node is polled. Based on this information, the collector adjusts the rate at which it performs polling cycles, also referred to as polling rate (but recall that each cycle may include several consecutive polls to empty the buffers of sensor nodes). The strategy of the collector is intentionally kept simple, namely to set the polling rate equal to the highest data rate reported by a sensor node. We note that this strategy is suboptimal from a latency perspective, as a data item will remain in the buffer of a sensor node for half the polling period on average, and for an entire polling period in the worst case. Since the objective of our experiments is to evaluate the ability of the protocol to adapt the polling rate according to the average application data rate, we do not try to optimize latency.

Unknown but stable data rates

In a first set of experiments, we evaluate the ability of the polling protocol to track a stable but unknown data rate of the sensor nodes. We run the rate estimation heuristic with the filter reset logic disabled (no-reset version). We have experimented with different application data rates and numbers of sensor nodes, for periodic as well as stochastic (Poisson) data generation scenarios. We present indicative results obtained for 0.5 data packets/sec and 3 nodes (other configurations result in similar behavior). The results for periodic and stochastic data generation are shown in Figure 6.4 and Figure 6.5, respectively The top plot in these figures shows the actual application data rate of each sensor node and the polling rate of the collector. The bottom plot shows the total number of polls and void polls that are performed by the collector To better illustrate the system dynamics, we plot the mean of the recorded values, calculated over a window of 10 seconds, over the entire duration of the experiment.

In the periodic data generation experiment, after a short warm-up phase, the sensor nodes accurately estimate their local application rate. As a consequence, the collector per-
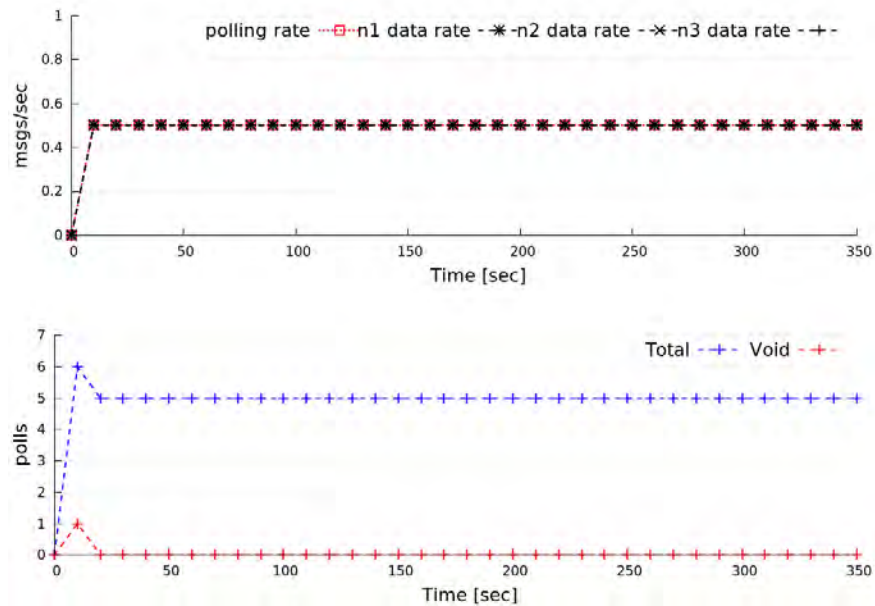
Figure 6.4: Periodic data generation with an average rate of 0.5 data packets/second and 3 sensor nodes.
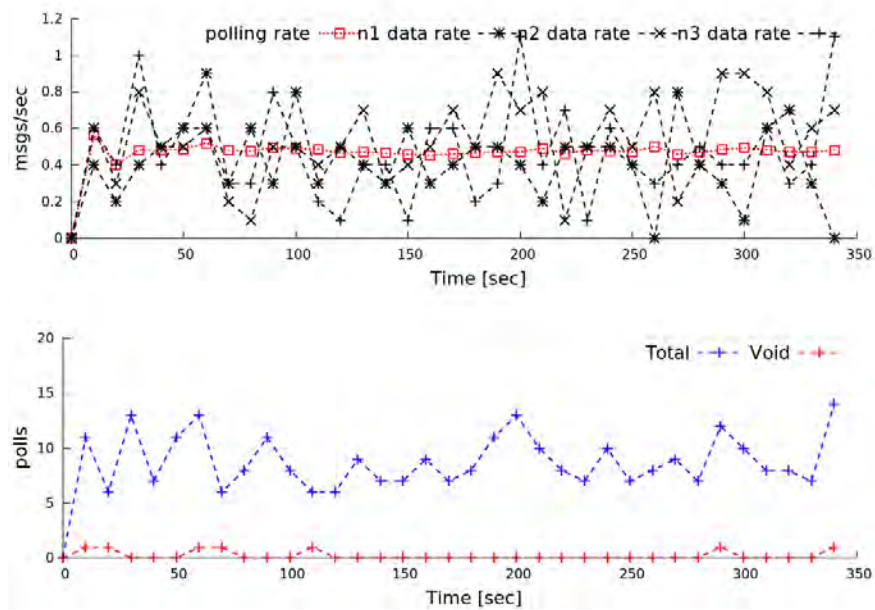

Figure 6.5: Stochastic data generation with an average rate of 0.5 data packets/second and 3 sensor nodes.
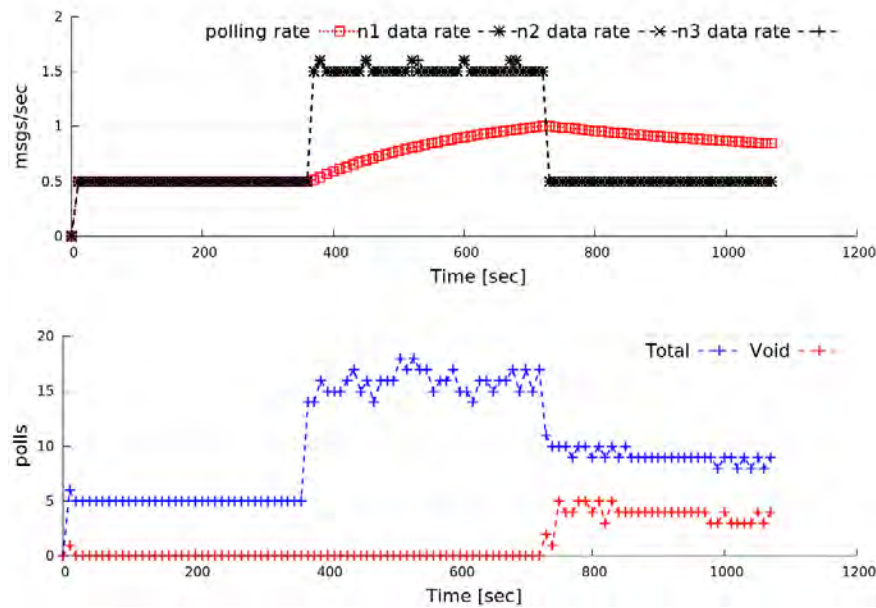
Figure 6.6: Periodic data generation with dynamic rate transitions, filter reset disabled.

forms polling cycles at exactly the right rate, avoiding any void polls. The same observation regarding the accuracy of the estimated data rate also applies to the stochastic data generation scenario. However, in this case, due to the inherent variance of the application data generation process, some sensor nodes end-up having several data items in their buffer, and the collector often has to perform several polls per cycle to retrieve this data. The variance in data generation also results in the opposite effect, namely it can be that no sensor node generates any data within the next polling interval, in which case the collector will perform a void poll. But, this happens quite rarely.

Unknown and dynamically changing data rates

In a second set of experiments, we evaluate the ability of the polling protocol to track dynamic transitions between significantly different unknown data rates of the application. Here, we test the rate estimation heuristic with the filter reset logic disabled and enabled (no-reset vs. reset version). We investigate a scenario where the sensor nodes initially have a data rate of 0.5 data packets/sec, then switch to a data rate of 1.5 data packets/sec, and after some time switch back to the initial data rate. As an indicative configuration, we present the results for 3 nodes, for periodic as well as for stochastic data generation (the results are similar for other configurations).

The results for the periodic data generation scenario are shown in Figure 6.6 and Figure 6.7. The no-reset version performs well in the first phase of the experiment, where the data rate does not change. However, when the application switches to a different rate, the
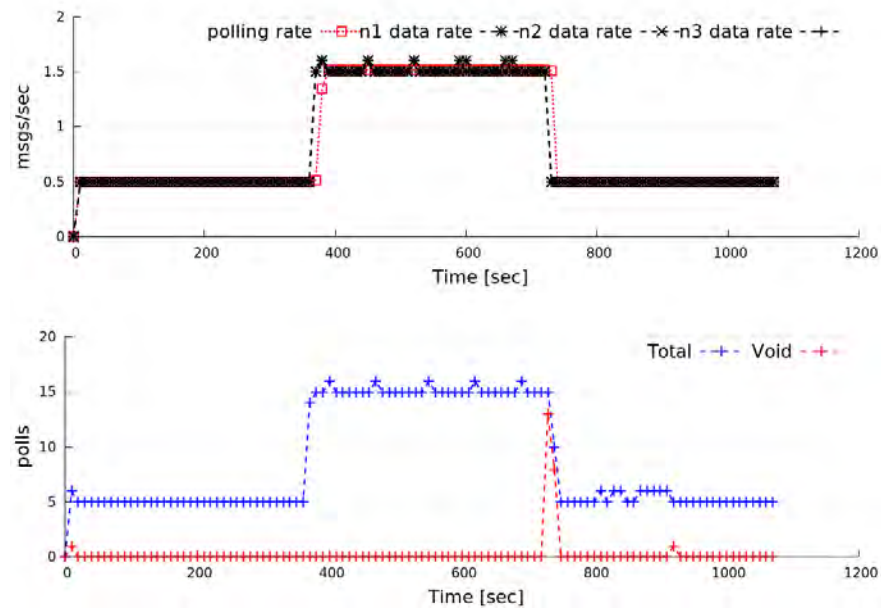
Figure 6.7: Periodic data generation with dynamic rate transitions, filter reset enabled.

estimation filter reacts slowly and is unable to find the actual data rate. The inability to track the first rate switch results in infrequent polling cycles, which leads to an increased number of polls. Conversely, after the second switch, inaccurate data rate estimation results in overly frequent polling, and to a significant number of void polls.

In contrast, the reset version tracks the rate transitions of the application smoothly and accurately. Thus the collector always polls the sensor nodes at the right rate. There are just a few void polls when the application makes the transition from the higher to the lower data rate, at which point the filter is reset and can immediately track the new data rate.

The results for stochastic data generation are shown in Figure 6.8 and Figure 6.9. Again, the no-reset version fails to track the rate changes of the application. The reset version is better in this respect, but in this particular case this does not always translate in better behaviour. Namely, due to the variance of data generation, the reset version adjusts the polling rate in a jerky way, even when the average rate is stable, and performs more void polls than the no-reset version in the first and second phase of the experiment. In the third phase, when the application switches from a high to a low rate, the more accurate estimation of the average data rate pays-off, leading to a reduced number of polls and void polls compared to the no-reset version.
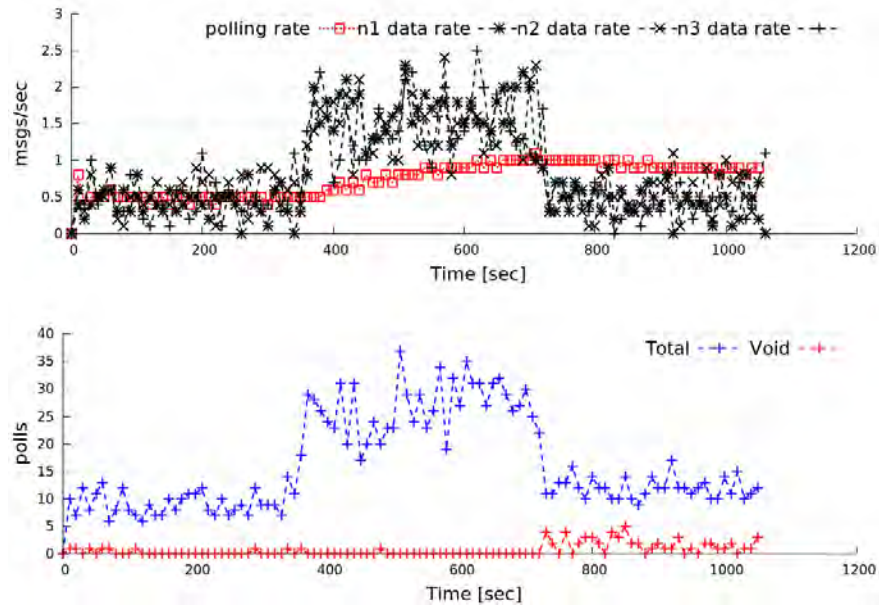
Figure 6.8: Stochastic data generation with dynamic rate transitions, filter reset disabled..
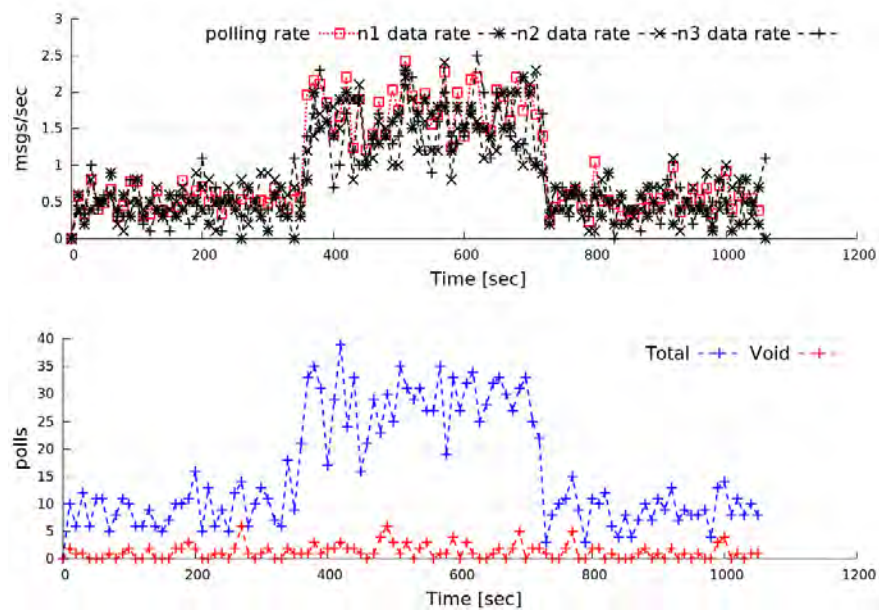

Figure 6.9: Stochastic data generation with dynamic rate transitions, filter reset enabled.

6.5 Related work

The efficient data transfer in networks where the channel capacity is limited has been studied at different layers of the network stack.

S-MAC [104] uses CSMA to eliminate packet collisions, and on top of that implements a relaxed time synchronization among nodes using a fixed sleep-wakeup schedule for nodes to transmit their data. T-MAC [100] follows a similar approach, but it employs an adaptive logic for the sleep-wakeup schedules based on a timer which is adjusted based on the network load. Both approaches are designed to be energy efficient, however they do not perform well for high network loads.

Z-MAC [89] is adaptive to the network load. It uses CSMA for low network loads, and switches dynamically to a version of TDMA when high network load is sensed. In the TDMA mode, nodes can use empty slots based on a priority scheme where the owners of the slots have higher priority than others. Z-MAC achieves high throughput, but it requires clock synchronization among the senders, and needs extra carrier sensing for the utilization of empty slots.

Another protocol that tolerates variations in periodic network load of a sensor network is SCP-MAC [105]. In this case, the nodes are sleeping and periodically wakeup to perform channel polling. When a node has data to transmit to a destination, it first sends a short preamble to the respective receiver, which in turn is activated to receive the data. The nodes are strictly synchronized in a fixed schedule in order to wakeup for the preambles. Prior to the transmission of the preamble, the medium is sensed to avoid collisions between multiple senders. A more adaptive approach is adopted in WiseMAC [49], where each node learns the sampling period of its neighbours, and creates a local wakeup schedule. The local schedule of a node is transmitted to the receiver in every data transmission. The receiver combines its own schedule to the one received, to minimize void wakeups (being ready for packet reception even though no node will transmit).

In terms of reliability, a number of protocols have been proposed above the MAC layer. In RCRT [82] several reliable unicast flows are directed from the nodes to a sink, which is responsible for controlling their data rate. The protocol guarantees reliability as the missing packets are recovered through a negative acknowledgement scheme. However, as the network size increases the supported transmission rates drop significantly due to high contention. RBC [106] also uses negative acknowledgements, but takes a different approach to decrease channel contention by giving priority to the nodes that have more data to transmit. The protocol employs an implicit blocking mechanism where each data transmission piggybacks the buffer size of the node. Senders overhear the transmissions of other nodes and block for a period of time if the overheard buffer size exceeds a certain threshold. This technique avoids contention, but nodes with a higher data rate might lead to the starvation of nodes with lower rates.

Unlike the above protocols, the polling approach based on GCBRR can support high data rates close to channel capacity with high reliability and thus practically zero retransmissions, without relying on a specific wireless technology. Also note that our approach is fair for all

nodes, and can be easily extended to support priorities without starvation by adjusting the mask used to specify which processes should reply; note that this is already done when performing consecutive polls in order to target nodes that have a non-empty data buffer. It can also be applied on top of very simple radios, and does not require clocks or some elaborate synchronization between the transmitting nodes. Finally, thanks to the ability of the protocol to dynamically adapt the polling rate to the actual data rate of the nodes, the number of unnecessary/void poll rounds can be significantly reduced when the application switches to low data rates.

Chapter 7

Simulation environment

This chapter presents the simulation environment which we have developed to support the testing of TeCoLa software stack and to allow easy experimentation with different mission programs and execution scenarios. We begin by providing some motivation for the simulation environment. Then, we present its architecture and proceed to discuss the implementation in more detail. Finally, we evaluate and verify the functionality of the implementation through indiactive mission scenarios.

7.1 Motivation

Performing experiments with unmanned aerial drones is challenging, even in the controlled setting of a testbed. First of all, extensive physical involvement is required at the testbed site, namely: (i) before the mission starts, the drone has to be prepped and brought to the default starting position or take-off point; (ii) during the mission, the drone has to be watched-over closely for safety reasons; (iii) after the mission, the drone has to be collected, maintained and stowed away until the next experiment takes place. Further, in part due to the above overheads, a drone testbed may not be available at all times, and the user may have to wait for a long time before acquiring a slot for conducting his/her experiment. This in turn significantly increases the development cycle of missions. Also, it is practically hard or impossible for the user to perform experiments for a wide range of missions, different weather conditions and different sensor inputs let alone in a reproducible way. Things become harder still, when considering collaborative missions that involve multi-drone scenarios. Last but not least, a drone experimentation testbed has a non-negligible running cost in terms of equipment maintenance and personnel effort.

From a functionality perspective modern drones can be considered as mobile wireless sensor/actuator platforms like those that have been described in the rich literature of wireless sensor networks (WSNs). Consequently, the development and testing procedures of missions that target drone-based systems have many similarities with the ones used for the applications in WSNs.

For traditional WSNs, the development cycle of an application typically involves the use of a simulator that mimics the execution of the application by simulating the nodes hardware, the wireless medium for a given network topology, and the on-board sensors used by the application. But usually the nodes are assumed to be static or follow rather simple mobility patterns, without considering any dynamic side-effects of the physical environment. In addition, the different elements of the system, such as the nodes and the network, are tightly integrated within a single simulation engine with focus on specific hardware platforms and network technologies, which are hard to extend in order to meet the needs of complicated and heterogeneous system elements such as drones.

Simulation technologies have already been developed separately for different aspects of a drone-based mission scenarios. For instance, simulators like [102] can faithfully reproduce the dynamics of an aerial vehicle using a suitable model that captures the physical behaviour of the vehicle when various forces are being applied to it. The model is then tightly-coupled with the autopilot subsystem, which runs on a real or simulated hardware platform. Such a setup can be used to test the effectiveness of low-level control algorithms/loops for driving the vehicles motors and steering elements, as well as for higher-level navigation logic. Also, mature network simulation engines like NS3 are capable of accurately simulating the behaviour of popular wireless protocols, such as WiFi and WiMax, which are used by modern drones to communicate with each other but also with external systems. Finally, virtual sensor mechanisms such as [68] can provide the application with real or synthetic data, allowing to explore a wide variety of scenarios.

In our work, we combine such state of the art simulation technologies into a unified and modular simulation environment, called AeroLoop. AeroLoop can be used to experiment with multi-drone mission scenarios in a flexile, efficient, cheap and safe way, without the bureaucratic burden, the very significant cost and the real risks of physical experimentation. In particular, it is possible to test middleware such as TeCoLa as well as mission-level software, before conducting trials in the field. This can be done without performing any simulator-specific adjustments in the software thereby allowing the software that has been tested in the simulator environment to be transferred to the real platforms without any modifications. Importantly, AeroLoop can be extended by incorporating third-party simulation components according to experiment needs.

7.2 System Architecture

Figure 7.1 shows the high-level system architecture of AeroLoop. The main entities are:

(1) The virtual unmanned aerial vehicles (vUAVs) that represent simulated autonomous aerial vehicles capable of performing a flight in the context of a mission.

(2) The virtual ground control station (vGCS) that runs the mission controller through which vUAVs can receive mission-related commands.
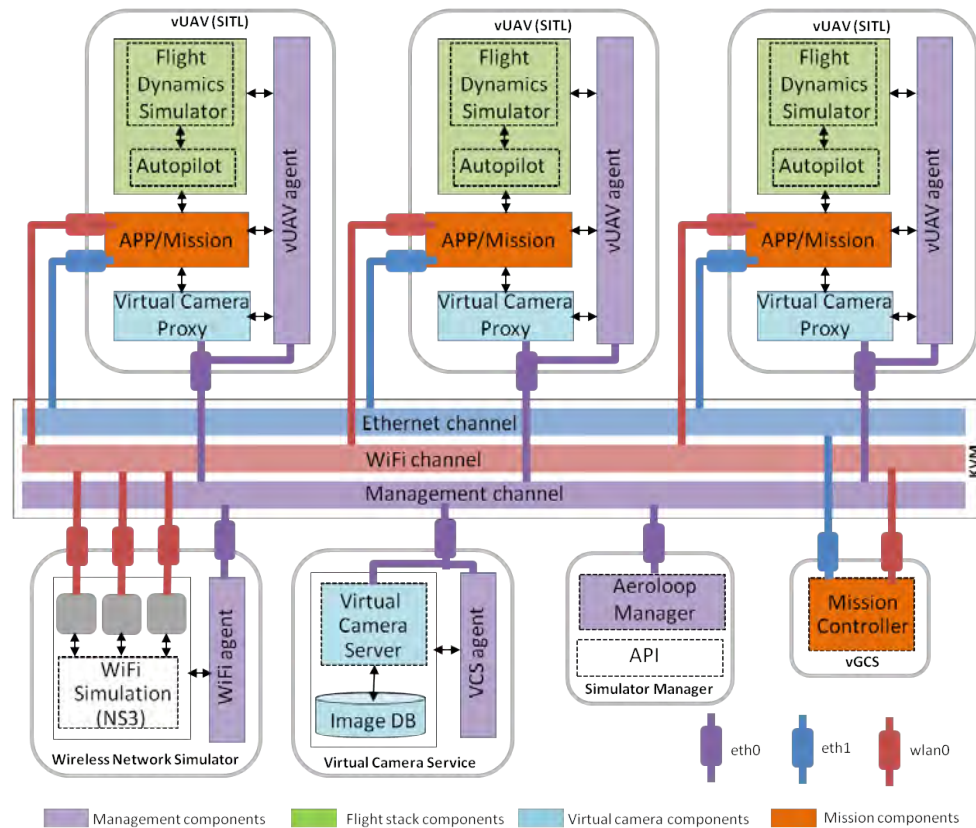
Figure 7.1: Architecture of the AeroLoop simulation environment.

(3) The wireless network simulator that is responsible for simulating the wireless communication among the vUAVs and between the vUAVs and the vGCS.

(4) The virtual camera service (VCS) that is responsible for serving aerial images of the mission area to the vUAVs.

(5) The AeroLoop manager that is used to perform the necessary initialization, configuration and control of all the previous system entities, through suitable interaction with the management components/agents within each such entity.

For better modularity and flexibility, each of these system entities is packaged as a separate Linux-based Virtual Machine (VM) that runs on top of the KVM hypervisor [7]. Each VM integrates all the software components that are required for the implementation of the respective functionality as well as for the interconnection and management of these entities.

The system-internal data exchange and management interactions between the AeroLoop manager and the management agents of the vUAV, WiFi and VCS entities, are performed via the dedicated management channel of the AeroLoop system. This is implemented using the native virtual networking facility of KVM, and is accessed within each system entity via

the wired network interface *eth0*. The desired control and information exchange is achieved using two different mechanisms. On the one hand, the AeroLoop manager issues control commands to the vUAV, WiFi and VCS agents via RPCs using the zerorpc framework [8]. On the other hand, the position updates of the vUAVs are propagated within the AeroLoop system via a pub/sub scheme using the zeromq framework [9]. More specifically, each vUAV agent periodically publishes the vehicle's position, and the WiFi agent and the AeroLoop manager both subscribe for such position messages in order to update their internal state accordingly.

vUAVs can communicate with each other and with the Ground Control Station using the WiFi and Ethernet channels. The WiFi channel is implemented using the NS3 [10] simulation environment, and is accessed within the VMs via the wireless network interface *wlan0*. This is intended to be used for the ad-hoc wireless communication among the vUAVs. Optionally, it can also be used for the communication between the vUAVs and the vGCS. The Ethernet channel is primarily intended for the communication between the vUAVs and the vGCS. Like the management channel, it is implemented using the native virtual networking facility of KVM, and is accessed within the VMs via the wired network interface *eth1*. Compared to the WiFi channel, the Ethernet channel allows for a more reliable communication between the vUAVs and the vGCS, and can be used to mimic more robust telecommunication/telemetry links that do not suffer any interference or collisions from the ad-hoc WiFi communication that takes place among the vUAVs.

7.3   Implementation

Virtual unmanned aerial vehicle (vUAV)

The vUAVs currently simulate quadcopters in a software-in-the-loop (SITL) configuration. Each vUAV is implemented using a Linux VM that runs the flight software stack, the on-board application logic, and the vUAV agent.

The flight software stack includes the flight-dynamics simulation engine for the quad-copter and the APM autopilot [2]. The flight-dynamics engine implements the physics model of the vehicle and defines the three-dimensional movement of the quadcopter under the forces and moments applied to it from various control mechanisms and the environment. It pro-duces artificial data for all flight-related sensors including GPS, which are fed to the autopilot in the same way as if this data were sensed on board during a real flight. Based on this data, the autopilot decides the actions that need to be performed in order to implement the navigation operations according to the experimentation scenario, and the respective control outputs are fed back to the flight simulation engine, closing the loop.

This setup makes it possible to test and verify the functional properties of the autopilot and its ability to stabilize the vehicle for a wide range of operation modes, including takeoff, cruise and landing. Following the trend of the APM community, we employ a recent version of the APM autopilot that implements control operations via a hardware-independent software

layer in order to support portability across different platforms. The flight simulation engine appears as a special platform, for which we use the publicly released APM port [11]. The execution of the autopilot drives the execution of the flight simulation engine in a lock-step fashion, eliminating the need for a more complex time synchronization between them.

In terms of connectivity, as already mentioned, the vUAV features three virtual network interfaces, one for each communication channel of the AeroLoop system, as illustrated in Figure 7.1. More specifically: (i) a wired interface (*eth0*) for the internal system management channel, (ii) a wireless interface (*wlan0*) for the ad-hoc communication between vUAVs via the simulated ad-hoc WiFi channel, and (iii) a wired interface (*eth1*) for the communication with the vGCS via the Ethernet channel.

In terms of mission-level sensors, besides the GPS provided by the flight simulation engine, vUAVs feature a virtual camera sensor device. For convenience, this is accessed through a high-level API, which, in turn, communicates behind the scenes with a system-wide virtual camera sensor service in order to retrieve real aerial photos based on the current position of the vUAV.

The vUAvs can also run one or more applications on-board. These can be used to enhance the autopilot functionality by adding more intelligence to the flight behavior of the vehicle or can be used to instruct the autopilot to perform a mission. They can also be used to perform sensor data processing tasks, or to implement more advanced functions, such as object recognition/tracking, path planning and 3D modelling, directly on board.

Finally, the vUAV agent is an AeroLoop-specific service that accepts commands from the AeroLoop manager and performs the corresponding initialization/start/stop operations of the flight stack. Furthermore, during simulation, the agent retrieves the GPS position of the vUAV from the autopilot, and forwards it (through the management channel) to the AeroLoop components that have an interest for this information.

Vehicle control/management

The flight of a vUAV is driven by control commands sent to the autopilot. This can be done in two ways/modes: (i) locally, based on a pre-programmed mission scenario or an application that runs on the vUAV and issues navigation commands to the on-board flight controller; (ii) remotely, based on commands sent through the vGCS.

The local mode utilizes the DroneKit-Python framework [12] that provides an API for communicating with the vehicles autopilot over MAVLink [13]. This allows to access the vehicles telemetry, state and parameter information, and enables both mission management and direct control over vehicle movement and operations. The remote mode utilizes the widely used MAVProxy ground control station [14], which is a consolebased application and provides a command-line interface for sending MAVLink commands to the remote vehicle.
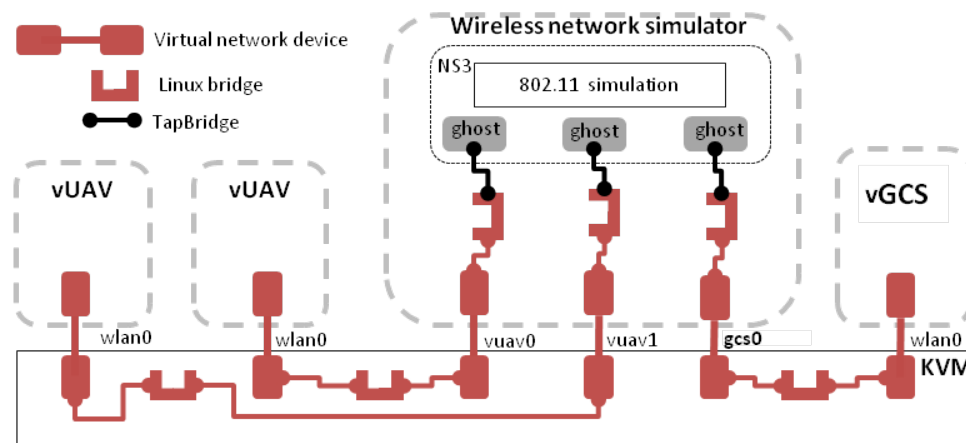
Figure 7.2: Organization of the simulated wireless network.

Simulated wireless network

The wireless network simulator is a Linux VM that implements the virtual WiFi network used for the communication between the vUAVs and the Ground Control Station. It consists of the NS3 network simulator, and the WiFi agent.

NS3 creates a network of communicating virtual nodes (so-called ghost nodes) and provides the modelling of IEEE 802.11 protocol standard at the physical and MAC level including the node mobility models. Each ghost node is connected to a properly configured system-level TUN/TAP interface, via the TapBridge mechanism of NS3 (which was suitably modified for our purposes). In turn, each such interface connects to an external system entity through a chain of virtual network devices and bridges via the KVM environment. As a result, the application software running within the vUAV as well as the mission controller in the vGCS can access the simulated wireless network via their local wireless LAN interfaces (*wlan0*), in a transparent way.

Figure 7.2 illustrates the setup behind the scenes, for a wireless network configuration with two vUAVs and the vGCS. In this case, the network simulator VM is configured to create three ghost hosts, which are bridged via TapBridge to three TUN/TAP virtual network devices (*vuav0*, *vuav1* and *gcs0*) that correspond to the three external system entities. In turn, each of these virtual network devices connects, through a bridge within KVM, to another virtual device that corresponds to the *wlan0* wireless network interface of the corresponding system entity.

Similar to the vUAV agent, the WiFi agent is an AeroLoop-specific service, which performs the initialization as well as the start and stop operation of the network simulation, based on the control commands received from the AeroLoop manager. During simulation, the WiFi agent receives from the vUAVs their current position, and updates the positions of the corresponding ghost nodes in order to adapt the characteristics of the wireless communication.
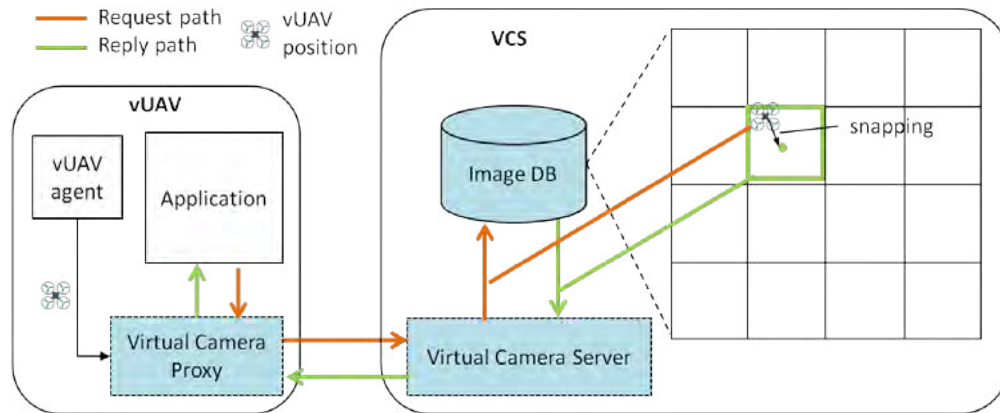
Figure 7.3: Implementation of the virtual camera sensor service.

Virtual camera support

The virtual camera sensor provides the functionality of a still snapshot camera, allowing the application running on the vUAV to take vertical aerial photos from its current position. The implementation follows the approach described in [68], and is illustrated in Figure 7.3. It consists of a Virtual Camera Service (VCS) that implements the key functionality, and the Virtual Camera Proxy (VCP) through which the VCS is accessed from within a vUAV in a transparent way. The VCS is a separate system entity running within its own VM on top of KVM, whereas the VCP component resides within the VM of each vUAV. In the general case, several vUAVs may access the VCS concurrently through the respective VCPs.

The VCS consists of a server program that handles the requests issued from the VCPs, and a collection of aerial images for the area of interest. The images are stored as separate files in the local file system. Upon instantiation, the VCS server creates an in-memory index structure that holds the geographical coordinates (latitude, longitude and altitude) and the file name for each image in the collection. During operation, the server handles requests coming from the VCPs, and returns the corresponding replies. Similar to the vUAV and WiFi entities, the VCS has a local management agent, which starts/stops and configures the VCS based on the commands received from the AeroLoop manager.

The VCP is part of the vUAV software stack. It represents a camera module attached to the vUAV, and provides a high-level interface to the application for retrieving the properties of the camera sensor and capturing still images. As part of its initialization, the VCP retrieves the contact details of the (remote) VCS from the local vUAV agent, and continuously retrieves position updates from the flight simulation engine at runtime. Upon invocation, it sends a corresponding request to the VCS, and stores the received image to the directory specified by the application.

When the VCS server receives a snapshot request, it searches the index to find and return the image whose center is closest to the 3-D position of the UAV (latitude, longitude, altitude). In other words, the image returned to the VCP is an original aerial image from the

collection, which corresponds to the position of the vUAV snapped to the closest data point of the 3-D index. This is illustrated in Figure 3, for the simple case of a 2-D index, where the image collection covers a rectangular area with images taken at a certain altitude in a grid-like manner with no overlap between them. In the general case, the image collection will cover multiple such horizontal planes at different altitudes, and each plane will be covered through numerous images with some overlap between them.

The VCS and VCP are programmed in Python. The remote interaction between them is achieved using the Python Remote Objects framework Pyro4 [15], with the respective communication taking place over the management channel of the AeroLoop system. On each side, the initialization/configuration of the Pyro framework, including the selection of the serialization method and the registration/binding process, is performed by the respective management agents. The VCP API is a simplified version of the *picamera* interface [16] of the Raspberry Pi application development framework.

System management

The notion of system management in the AeroLoop system has some analogy to the preparations that have to be done in a real UAV testbed, before an experiment can take place. As already discussed, system management is performed through the AeroLoop manager, which in turn interacts with the management agents of all other system entities.

The AeroLoop manager provides a structured API through which it is possible to perform the basic management operations and receive information about the state of the system. The API is implemented using the zerorpc technology [8] so that it can be conveniently invoked in a platform-neutral way from any external/remote system.

The API supports the setup/configuration of the different entities of the AeroLoop system in order to perform an experiment. Note that one may also inspect the state of the vUAVs, the WiFi simulator, the VCS and the vGCS without going through the AeroLoop manager, by logging into the corresponding VM via *ssh* to issue control commands and/or inspect the log files of different programs in a direct way.

7.4   Evaluation

To verify the functionality of our implementation we use an indicative configuration that consist of two vUAVs, the VCS, the WiFi simulator and the vGCS. Before an experiment, the simulator environment needs to be initialized and the various system entities need to be properly configured and started. The initialization is performed through a startup script that uses the API of the AeroLoop manager. Once the simulation environment is properly configured, the mission experiment. In ordert to test each simulation entity individually we use mission scripts that run locally on the vUAVs and instruct the autopilot directly.

Setting up an experiment

Before running an experiment, the simulator environment needs to be initialized, and the various system entities need to be properly configured and started. Listing 7.1 shows the relevant code excerpt of a startup script that uses the API of the AeroLoop manager to setup an indicative configuration with two vUAVs, the VCS, the WiFi simulator and the vGCS.

Listing 7.1: Experiment configuration/startup procedure.

```python
1  # Create AeroLoop manager proxy object
2  aeroloop = zerorpc.Client()
3  aeroloop.connect(CONNECTION_STRING)
4  # Register simulator entities
5  aeroloop.register_vm(VUAV_A, "vuav")
6  aeroloop.register_vm(VUAV_B, "vuav")
7  aeroloop.register_vm(WNS_VM, "wifinet")
8  aeroloop.register_vm(VCS_VM, "camsrvc")
9  aeroloop.register_vm(GCS_VM, "gcs")
10 # Start simulator entities
11 aeroloop.start_vm(VUAV_A)
12 aeroloop.start_vm(VUAV_B)
13 aeroloop.start_vm(WNS_VM)
14 aeroloop.start_vm(VCS_VM)
15 aeroloop.start_vm(GCS_VM)
16 # Configure wireless network
17 aeroloop.create_wifi_segment()
18 aeroloop.create_ghost_node(VUAV_A)
19 aeroloop.create_ghost_node(VUAV_B)
20 aeroloop.create_ghost_node(GCS_VM)
21 # Set home positions
22 aeroloop.set_vuav_home_pos(VUAV_A, HOME_POS)
23 aeroloop.set_vuav_home_pos(VUAV_B, HOME_POS)
24 aeroloop.set_vuav_home_pos(GCS_VM, HOME_POS)
25
26 aeroloop.set_ghost_node_pos(VUAV_A, HOME_POS)
27 aeroloop.set_ghost_node_pos(VUAV_B, HOME_POS)
28 aeroloop.set_ghost_node_pos(GCS_VM, HOME_POS)
29 # Start simulation engines/services
30 aeroloop.start_vuav(VUAV_A)
31 aeroloop.start_vuav(VUAV_B)
32 aeroloop.start_wifi_ns3_sim()
33 aeroloop.start_vcs(URBAN_IMAGES_COLLECTION)
```

In a nutshell the steps followed are: create an AeroLoop manager proxy object (lines 2-3); register the simulator entities with the manager (lines 5-9); start the simulator entities (lines 11-15); create a wireless broadcast domain (line 17), and a ghost node for each vUAV
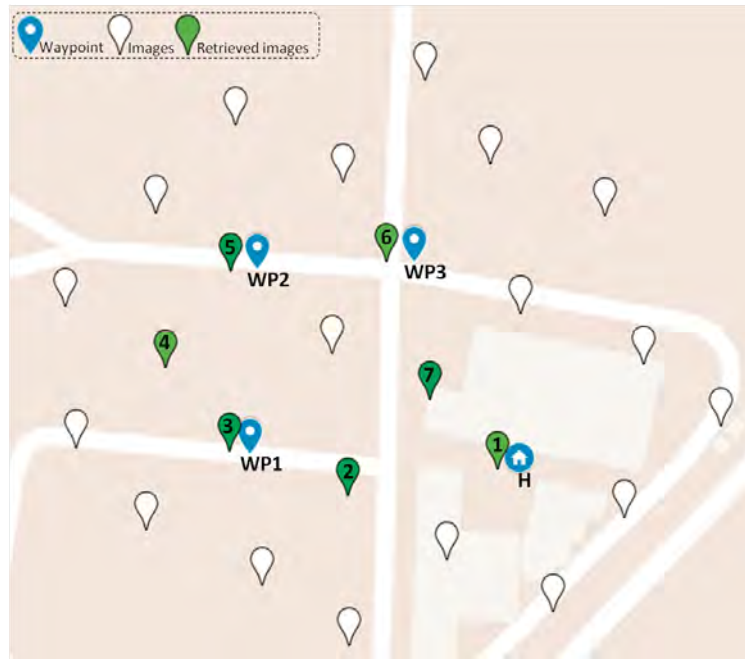
Figure 7.4: Mission area, waypoints and positions of the images in the collection used for the VCS (the filled markers represent the images that are actually retrieved during the mission).

and the vGCS (lines 18-20); set the home positions for the vUAVs/vGCS (lines 22-24) and the corresponding ghost nodes (lines 26-28); start the vUAV and WiFi simulation engines (lines 30-31 and 32, respectively); and finally start the VCS with the image collection for the mission area (line 33).

At this point the simulation environment is properly configured, and one can start a mission experiment, either by running a suitable application program directly on the vUAV, or by issuing commands from the vGCS.

Camera sensor

We test the virtual camera via a simple local mission script where a vUAV starts from a home position H, visits a sequence of waypoints (WP1, WP2, WP3) and returns to home. For this mission, the image collection of the VCS is populated with images taken from the *urban mapping* dataset that is publicly available from senseFly [17]. Figure 7.4 shows the mission area and waypoints, as well as the (center) position of the original aerial images in the collection used for the VCS.

After take-off, we *ssh* into the vUAV and start an application that takes pictures periodically, every 5 seconds. The vUAV hovers at position H for 5 seconds, then starts its trip in order to visit the waypoints, and when it returns to H it hovers there for another 5 seconds. The trip time is about 70 seconds, so the total mission is about 80 seconds in total.

Figure 7.5: Sequence of the images captured by the vUAV during the mission. The label at the top left corner of each image is the unique identifier of that image in the collection.

The photo sequence taken during the mission is shown in Figure 7.5 (since the mission is about 80 seconds and the application program invokes the virtual camera every 5 seconds, the vUAV takes 16 photos). We confirm that the images captured by the virtual camera sensor, denoted by the filled markers in Figure 7.4, are the closest ones to the flight path (a straight line between the respective waypoints). Also, some images appear more than once in the photo sequence (see the label of each image in Figure 7.5). This is due to the sparsity of the image collection for the mission area, which, in turn, leads to a relatively large snapping distance within the VCS. To minimize the snapping distance and let the application receive images that closely correspond to the actual vUAV position, the image collection should cover the area in question at a sufficient spatial density.

WiFi communication

In a second experiment, we test the WiFi simulation functionality. For this purpose, we use two mission scripts in two vUAVs, and employ the *iperf* utility to measure the throughput between them. The one vUAV performs the same trip as above, while the other simply hovers at position H. Before the experiment starts, we *ssh* into the vUAVs, and manually start an *iperf* server and an *iperf* client, respectively, which run in the background during the mission.
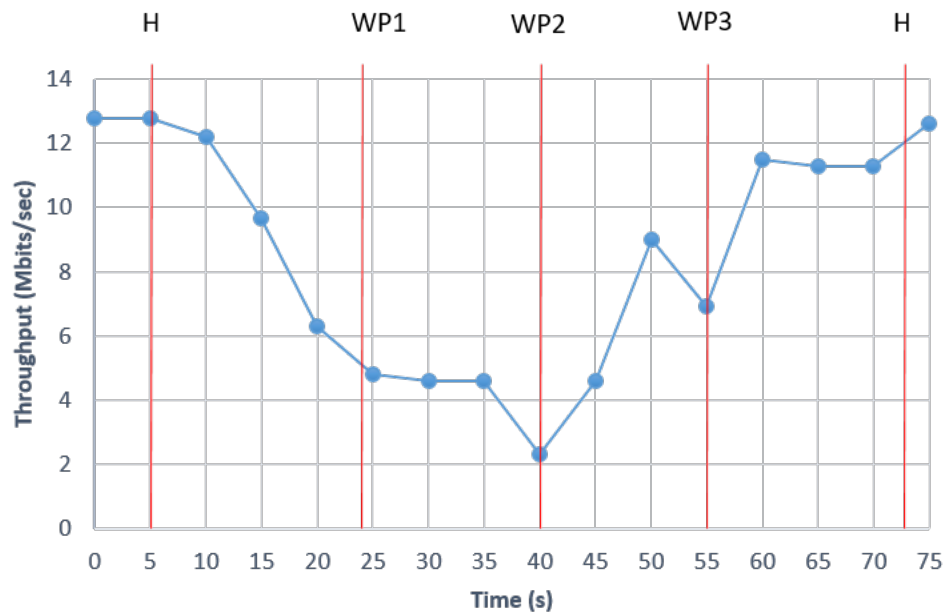
Figure 7.6: Throughput between the two vUAVs during the mission. The red vertical lines mark the points in time when the moving vUAV reaches the different waypoints.

Figure 7.6 shows the results. As expected, the maximum throughput (12.8 Mbits/sec) is achieved when both vUAVs are at position H, at the start and at the end of the mission. As the moving vUAV flies away from H on its way to WP1, throughput decreases as the distance between the two vUAVs increases. This trend continues as the vUAV flies towards WP2, at which point throughput reaches the minimum value reported for the entire mission (2.31 Mbits/sec). This is reasonable given that WP2 is indeed the most distant point from H where the other vUAV is located. Conversely, once the moving vUAV gets past WP2 and the distance with H decreases, throughput gradually increases again.

Wind conditions

In a thrid experiment, we test the effect of wind on the flight behaviour of the vUAV. We use a single vUAV that runs the same mission script as in the first experiment. When the vUAV starts moving from WP2 to WP3, we introduce, through the AeroLoop manager, a gust with a speed of 30 m/s coming from the south (180 degress from North) as shown in Listing 7.2. We keep the gust for 10 seconds, and then reset the wind conditions to default/normal.

Listing 7.2: Simulated wind configuration.

```
1
2 # Set simulated wind
3 aeroloop.set_vuav_wind_dir(VUAV_A, 180)
```
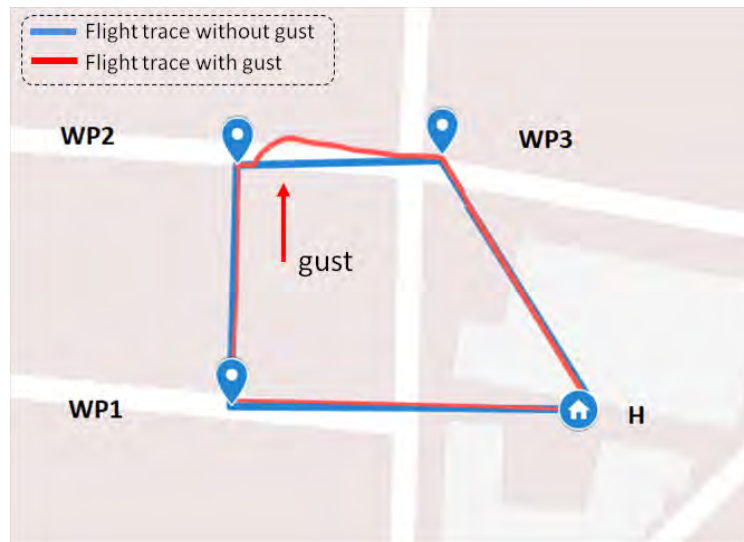
Figure 7.7: Flight traces of the vUAV, without and with the wind gust.

```
4  aeroloop.set_vuav_wind_spd(VUAV_A, 30)
5  ...
6  # Reset simulated wind settings
7  aeroloop.reset_vuav_wind(VUAV_A)
```

Figure 7.7 shows the flight trace compared to the one for the same mission without the gust. It can be seen that initially the gust pushes the vUAV away from its original course. The vUAV then tries to stabilize and adapt its path, and once the gust stops, it starts making progress and reaches the next waypoint WP3. This also has an impact on the flight time: it takes the vUAV 25 seconds to fly from WP2 to WP3, whereas without the gust this trip takes just 15 seconds.

Flight behaviour

In a final experiment, we compare the flight behaviour of the vUAV with a real UAV. Figure 7.8 shows the UAV which is a hexacopter of 60cm diameter with a payload lifting capability of up to 1200g. The UAV runs the APM autopilot software stack on top of the PX4 hardware platform and is also equipped with a RaspberryPI companion computer.

We performed our experiment with a simple mission script where the UAV starts for the home position H, visits waypoints WP1 and WP2 and returns to home position H. During the mission the UAV changes its altitude in various ways. In particular it performs a vertical take-off 20m above the home position H and maintains such altitude while it flies towards the waypoint WP1. Once the waypoint WP1 is reached the UAV increases its altitude to 30m and then returns to an altitude of 20m before starting its course to waypoint WP2. The altitude is increased again from 20m to 30m progressively while the UAV flies from waypoint
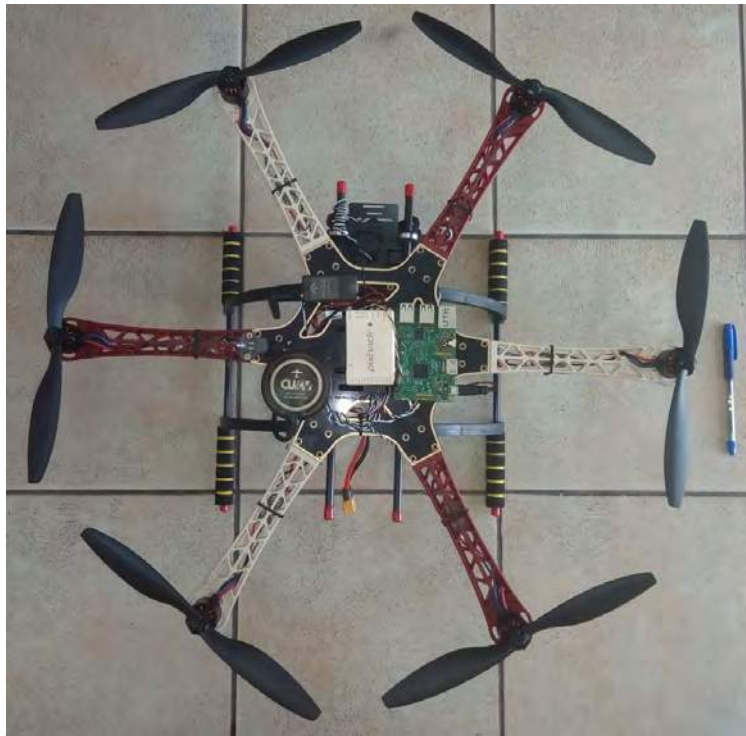
Figure 7.8: Hexacopter.

WP1 to waypoint WP2. Finally, the hexacopter returns to home position maintaining an altitude of 30m and performs the landing operation. The mission script was created with the mission planner [18] software. Figure 7.9 shows a snapshot of the mission planning.

We performed the mission above with a UAV in the field and with a vUAV in the AeroLoop simulator. Note that the vUAV simulates a quadcopter so our comparison focus on the high-level behaviour of the autopilot during the automated flight mode without comparing data from sensors and actuators.

Figure 7.10 shows the flight traces in 2-D and 3-D space of the vUAV and of the UAV based on data from the GPS. We observe that in both cases the vehicles follow the same path with an error of 2 meters on average. The most important differentiation occurs when the vehicles start to fly from waypoint WP1 to waypoint WP2 where a small wind gust cause the UAV to slightly divert from its lat/long track (6m) and from its altitude (3m). Such diversion is also illustrated in the altitude traces of Figure 7.11 where the altitude changes are shown in respect to time. In particular, we observe that the altitude of both vehicles change in the same way at the same time until the completion of the altitude decrement operation at waypoint WP1. When the UAV flies towards waypoint WP2 we observe that it takes more time to increase its altitude and to achieve the waypoint WP2 due to diversion in its course. Recall that the altitude is increased progressively while the vehicles approach the waypoint WP2 whilst the ascendant rate is calculated based on the distance between
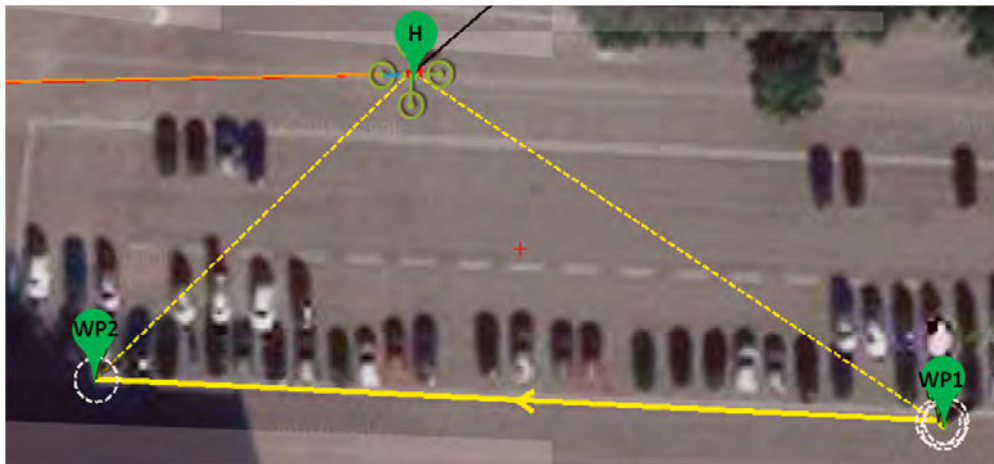
Figure 7.9: Mission planner snapshot of the mission planning.

the poisition of the vehicle and the the target waypoint. The presence of wind also cause the small altitude differentiation of 3m between the UAV and the vUAV when the vehicles return to their home position. Both missions have been completed in about 110 seconds however as it can be observed that the UAV completed the mission 3 seconds earlier than the vUAV despite the diversion in its course. This is because the UAV has a higher altitude descent rate during the landing operation than vUAV. During our experiment we didn't record the wind characteristics due to lack of instrumentation so it was not possible to perform a flight comparison with the presence of wind however our results show that vUAV simulation provides an accurate tool for experimentation.

7.5   Deploying and testing missions with TeCoLa

As already discussed, the AeroLoop simulation environment was used to test the TeCoLa software stack and the indicative mission scenarios described in Chapter 3 and Chapter 4. In the following, we briefly describe how TeCoLa is deployed on AeroLoop in order to enable such tests.

AeroLoop configuration

To support the testing of the forest fire detection and response mission program discussed in Section 3.4, we use one vGCS for the mission controller and four vUAVs. Two of the vUAVs represent the scanner drones and the other two the fire extinguishers. The vGCS and the vUAVs are configured to communicate through the virtual WiFi network.

For the crop spraying application in Section 4.7, we use two vGCSs for the replicas of the mission controllers, and three vUAVs for the sprayer drones. One vGCS serves as the primary, the other serves as the backup mission controller replica, which takes over when the
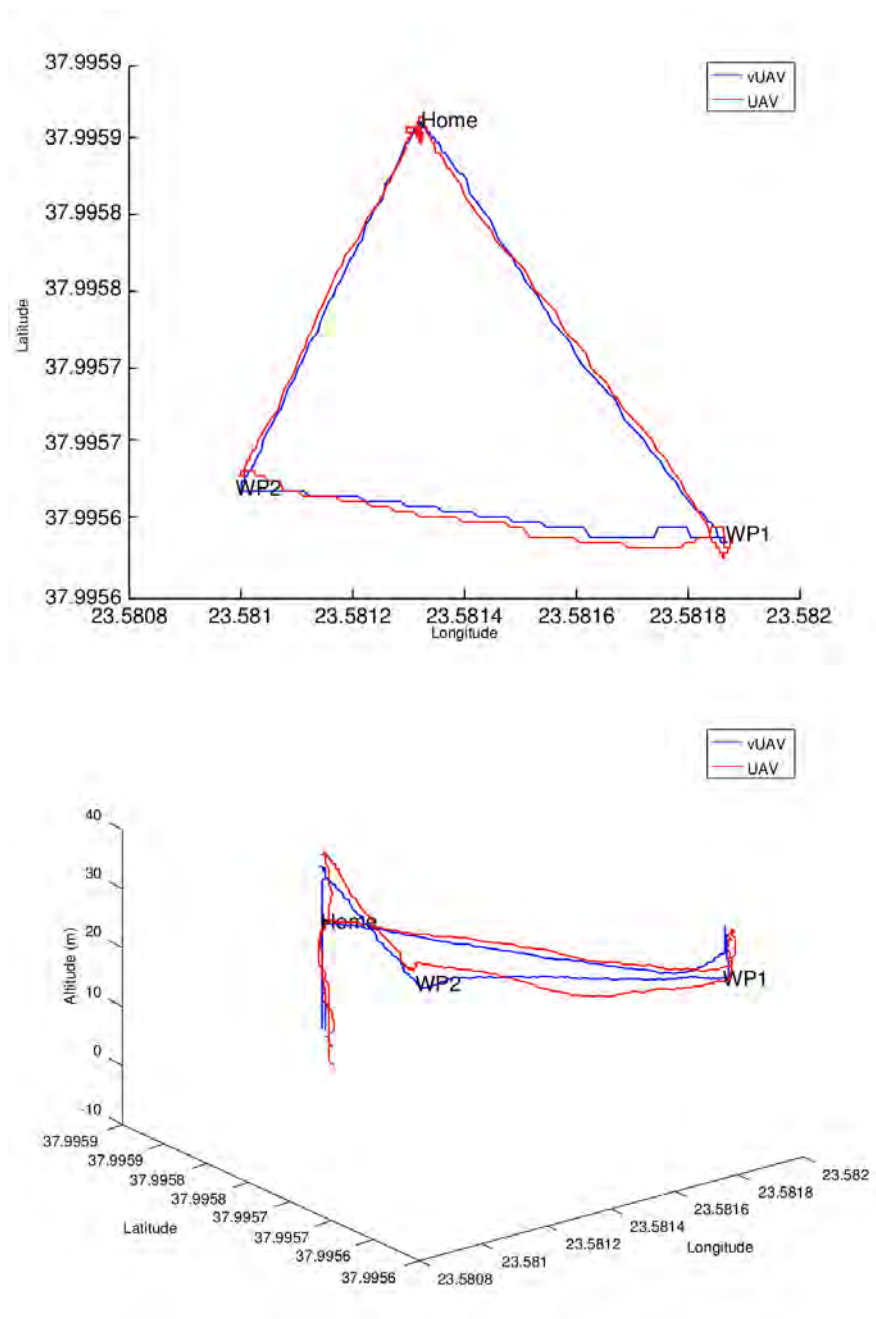
Figure 7.10: Flight traces in 2-D (top) and 3-D space (bottom) of the vUAV and the UAV.
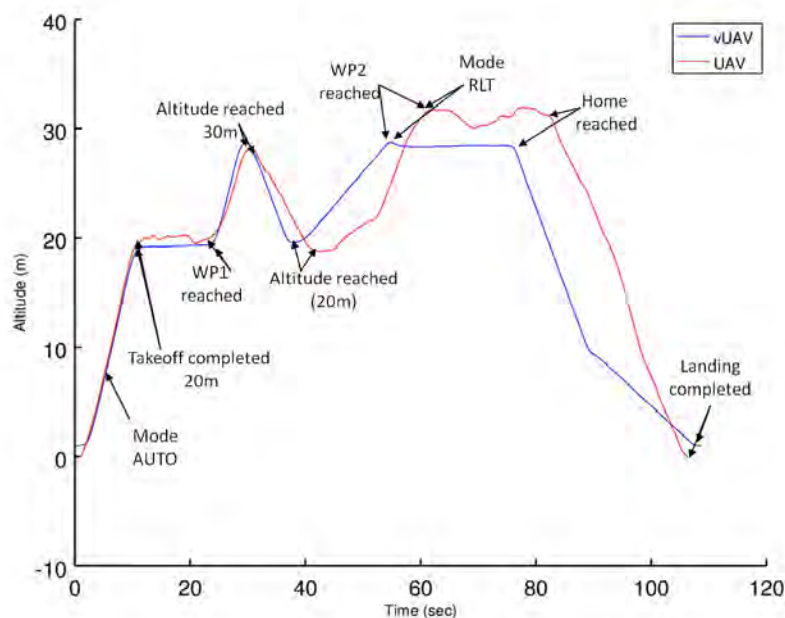
Figure 7.11: Altitude traces of the vUAV and the UAV.

primary fails (recall that these tests are performed to evaluate the tolerance to failures of the mission controller). As in the previous tests, the vGCSs and the vUAVs are configured to communicate through the virtual WiFi network.

TeCoLa deployment

We deploy the TeCoLa software stack in AeroLoop by installing the node runtime in the vUAVs and the mission controller runtime in the vGCSs. The transport layer of both runtime environments, which is implemented using the GCBRR protocol, performs the message exchanges through the wireless interfaces of the corresponding VMs.

Node services

In all experiments, the node runtime instantiates a real implementation of the mobility service. This uses the DroneKit-Python framework [12] over the MAVLink [13] protocol in order instruct the vehicle to perform the requested movements and to retrieve its current position. Note that this service implementation will also work on real UAVs that support the MAVLink protocol.

The rest of services used in the above experiments (photo service, fire detection service, fire extinguisher service, and field spray service) have dummy implementations that are pre-programmed to suit the respective test scenarios.

Faults

During out tests we can induce fail-stop crashes of the mission controller and/or vUAV nodes at any point in time, as needed to observe the system's behavior for different failure scenarios. To incude a fail-stop, we ssh into the corresponding VM and stop the respective process (mission control runtime or node runtime). As another option, it is possible to do this from within a program that runs on the VM. For instance, it is easy to write a program that retrieves state information from the logs, and stops the TeCoLa runtime process when certain conditions are met.

7.6 Related Work

In the field of unmanned/autonomous vehicles there are many works focused in the simulation development using different techniques and from different perspectives.

FVMS [92] is a a software-in-the-loop platform for the evaluation of control algorithms for a single quadrotor UAV. It uses the Microsoft Flight Simulator [19] for the modelling of the flight dynamics as well as the simulation visualization, and provides the emulation of embedded sensors and cameras. On the other hand, hardware-in-the-loop based simulation setups are focused on the testing for specific hardware platforms. For instance, MAV3DSim [78] targets the validation and verification of UAV controllers for the Pixhawk autopilot. Its simulation engine is based on the CRRCSim Simulator [20] for modelling the flight dynamics and visualizing the simulation, and provides a GUI for accessing the system variables/parameters similar to the Ground Control Station applications.

While the above examples target single-UAV simulations, computer-based simulators like UAVSim [63] and MultiUAV [88] support collaborative multi-vehicle missions based on software models of UAVs. UAVSim, which is built on top of the OMNeT++ simulator [101], is focused on security providing an accurate network simulation and a set of libraries for simulating various network attacks, whereas MultiUAV [88], which is built on Matlab/Simulink, targets the evaluation of cooperative control algorithms for multiple UAVs.

In terms of communication simulation, UAVSim [63] provides a network simulator for collaborative UAVs with focus on security. UAVSim is built on top of the OMNeT++ network simulator and provides a set of libraries for the simulation of various network attacks. Moreover, it incorporates software models of existing autopilots and support for swarm simulation scenarios. UAVsim provides an accurate network simulation that features wireless propagation models for multiple radio channels and MAC-level protocols, including the 802.11 MAC. It also provides detailed configuration and analysis for various types of network attacks however it does not provide support for mission-specific sesnors.

In [26] a software framework for simulation benchmarking of scenarios including multiple vehicles of different types is described. It is constructed over LabVIEW and each autonomous vehicle consists of three modules for modelling the dynamics, the navigation control and the communication disturbance. Each one of these modules can be defined by a

DEVS/non-DEVS model, a continuous model, or an interface that interacts with hardware or MATLAB/Simulink, therefore allowing modular hybrid HIL/SIL simulations.

Thetis [84] is another multi-vehicle simulator specialized for unmanned underwater vehicles (UUVs). It consists of four separate simulation engines for simulating the environment, the communications, the vehicle and the embedded sensors. Each UUV interacts with the vehicle simulator, the communication simulator and the sensor simulator. In turn, these simulators are hooked to the environment simulator which is in charge of providing the operation field.

From the above mentioned simulators AeroLoop has similarities mainly with Thetis which follows the same holistic simulation approach. However, Thetis is designed to support only UUVs as it is tightly coupled with the environment simulator. On the contrary, the modular architecture of AeroLoop allows the integration of various types of simulated vehicles, of simulated networking technonolgies and of virtual sensors. Note that the absence of a centralized simulation environment as the one used in Thetis impose some limitations to AeroLoop for experiments where the vehicles require a syncronization based on a common reference other than GPS.

Chapter 8

Conclusions

We envision UVs to become an integral part of next-generation computer-based applications. Thanks to the already impressive and ever increasing autonomy in terms of movement, navigation and obstacle avoidance, there is little need for external low-level control and micro-management. However, UVs still need to be coordinated at a higher level in order to become part of a greater system as well as to perform missions that involve multiple UVs. This is the focus of the work presented in this thesis. In the following, we summarize our achievements and point to some directions for future work.

Summary

Writing mission programs that can exploit several and different types of UVs is a daunting task. To address this issue we adopt a centralized coordination model in order to give the developer the convenient illusion of programming a single platform. We employ a master-slave approach where the UVs are abstracted as nodes (slaves) capable to gather information via sensors and to perform actions via actuators. The nodes are controlled by a distinguished entity the mission controller (master) that runs the mission logic.

Based on this coordination model we have proposed TeCoLa, a programming framework that simplifies the mission program development by offering suitable primitives and explicitly target UV heterogeneity. In TeCoLa, the nodes provide a number of services based of their hardware and software resources. Each service consists of one or more service calls which are used to issue commands to the nodes and to retrieve state information from them. Services can be remotely accessed from the mission controller via remote service calls in the spirit of remote procedure calls (RPCs). The mission controller maintains the mission group which contains information regarding the nodes that participate in the mission. The mission controller provides to the mission program primitives for the creation of teams of nodes according to the mission objectives. TeCoLa also allows the promotion of the services that are common to all members to the team level enabling a unified control both at the node level and at the team level. The complexity of team formation and team dynamics, due to the addition and removal of individual nodes at runtime, remains hidden behind the

scenes, while providing suitable hooks to the programmer for adapting the mission program behaviour as needed.

To tolerate failures of the mission controller in a transparent way, we have proposed the technique of *selective replay*. We use a novel combination of checkpointing, passive replication and logging, which is able to resume mission program execution in a consistent way taking into account system/environmental dynamics, with practically zero effort from the mission program developer. We also discussed a concrete implementation of the proposed approach as part of the TeCoLa programming framework and identified the key overhead components of the fault-handling mechanism, both in an analytic way and through an indicative mission scenarios. Our results show that the proposed approach can achieve fast mission recovery, even when going through a replay phase before resuming normal execution.

To support the TeCoLa programming model, we have introduced the GCBRR protocol that supports the coordinated 1-N request/reply interaction pattern and group management in broadcast-based wireless systems. The protocol eliminates contention and exploits the broadcast nature of the shared medium to minimize the number of message transmissions and latency. GCBRR does not rely on any advanced MAC features, and can work on top of different networking technologies. We presented an evaluation over 802.11 WiFi for scenarios where the system operates close to the network capacity. Our result show that the proposed protocol achieves significantly higher throughput, lower latency and better predictability than unicast-based point-to-point approaches. These results are particularly encouraging given that WiFi comes with a very effective flow-control mechanism and MAC-level support for unicasts.

We have also investigated how GCBRR can be used to support N-1 information flows via polling, for the more general scenario where multiple sensor processes wish to send data to a single collector process over a low-bandwidth wireless channel. A core component of our approach is an application-agnostic mechanism for estimating the actual data rate of each process, so that the collector can adjust the polling rate accordingly. This is based on a Kalman filter that tuned for stable data generation rates, and which is controlled via simple signals generated at runtime by the polling protocol. We experimentally evaluate an implementation of our protocol for different data traffic scenarios using real nodes that communicate with the collector over IEEE 802.15.4 radio, showing that the collector can successfully track the actual data rate for both periodic and stochastic data generation.

Finally, to make it possible to experiment with multi-drone applications in a cheap, safe and flexible way, as well as to test TeCoLa software stack for different mission programs and execution scenarios, we have developed the AeroLoop simulation environment. AeroLoop provides a unified and modular simulation environment for easy and safe experimentation with virtual UAVs. It combines state of the art simulation technologies for flight, sensing and networking into a unified and modular simulation environment that can be used to experiment with multi-drone mission scenarios without the risks and the costs of physical experimentation. Our design takes advantage of mature virtualization technology which makes possible to test middleware such as TeCoLa as well as mission-level software without performing any simulator-specific adjustments. This allows the software that has been

tested in the simulator environment to be transferred to the real platforms without any modifications Moreover AeroLoop can be extended by incorporating third-party simulation components according to experiment needs. The functionality of AeroLoop has been verified extensively through the mission scenarios of Chapters 3 and 4 and also through individual mission scripts.

Future work

While our work addresses several important problems of multi-drone applications, there are more issues that are worth pursuing in the future.

In terms of the programming model, one could enrich the syntax of team membership and wait/select expressions in order to allow for more advanced declarative functionality. It would also be useful to support specific team formation patterns in a more institutional manner, as a first-class aspect of TeCoLa, rather than through mission-level code templates that have to be hand-crafted by the developer. Furher, one could provide support along the lines of TeCoLa embedded within a programming language, via suitable extensions at the syntax and runtime level. Another interesting direction could be the integration of TeCoLa with stationary sensor/actuator infrastructures, so that these can be seamlessly exploited by the mission.

An important issue regarding the fault-tolerant aspect, is to investigate optimizations that can substantially reduce the size of the checkpoint images, taking into account specific properties and features of the TeCoLa runtime environment. Furthermore, it would be very interesting to investigate active replication techniques, which would eliminate the need for expensive checkpointing and would also make it possible to tolerate Byzantine failures, while studying the extra communication overhead due to the required synchronization between replicas of the mission controller.

In terms of transport protocols, it would be interesting to extent GCBRR to support the 1-N request/reply interaction over a multi-hop network topology, actively exploiting the opportunities for parallel transmissions while preserving its key properties regarding the network contention and group management. This would allow to support missions where the UVs can be far apart from each other. It is also important to investigate protocol extensions to tolerate network partitioning, and to see how to deal with this aspect at the level of application programming.

Regarding the simulation environment, we plan to exploit the modular nature of the AeroLoop in order to add different types of virtual vehicles, such as aircrafts and ground rovers. Moreover, we wish to support a larger number of virtual sensors that are popular in the robotics community, such as LIDAR and IR. We also plan to investigate more lightweight forms of virtualization and network simulation using containers and Mininet [74].

Last but not least, we wish to test TeCoLa with real UVs, in a suitably controlled/contained physical testbed. Such type of experimentation would allow us to observe the behaviour of drones in real flight formations where there is interference between them. These observation would help us to design abstractions to support team formation patterns and

also to implement better failsafe operations. Until now we didn't have the necessary hardware, the required space and the training to conduct experiments with real drones however, soon drone-testbed infrastructures like RAWFIE [21] will allow us to perform such type of experimentation.

Bibliography

[1] https://www.mobilicom.com/will-post-smartphone-era-bring-wearable-drones.

[2] http://ardupilot.org/ardupilot/index.html.

[3] http://nitlab.inf.uth.gr/NITlab/index.php/testbed/
     nitos-facility-architecture.

[4] http://hbict.sourceforge.net/about.html.

[5] http://nitlab.inf.uth.gr/NITlab/index.php/hardware/wireless-nodes/
     icarus-nodes.

[6] https://www.advanticsys.com/shop/mtmcm5000sma-p-23.html.

[7] https://www.linux-kvm.org/.

[8] http://www.zerorpc.io/.

[9] http://zeromq.org/.

[10] https://www.nsnam.org/.

[11] http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.

[12] http://python.dronekit.io/automodule.html.

[13] http://qgroundcontrol.org/mavlink/start.

[14] http://ardupilot.github.io/MAVProxy/html/index.html.

[15] https://pythonhosted.org/Pyro4/.

[16] http://picamera.readthedocs.io/.

[17] https://www.sensefly.com/drones/example-datasets.html/.

[18] http://ardupilot.org/planner/.

[19] https://www.microsoft.com/Products/Games/FSInsider/product/Pages/.

[20] `https://sourceforge.net/projects/crrcsim/`.

[21] `http://www.rawfie.eu`.

[22] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer Science Business Media, 2004.

[23] L. Alvisi and K. Marzullo. Message logging: pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[24] J. Ansel, K. Arya, and G. Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *International Symposium on Parallel and Distributed Processing*, pages 1–12, 2009.

[25] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *International Conference on Intelligent Robots and Systems*, pages 2794–2800, 2007.

[26] A. M. Astorga, D. Moreno-Salinas, D. C. Garca, and J. A. Almansa. Simulation benchmark for autonomous marine vehicles in labview. In *OCEANS*, pages 1–6, 2011.

[27] J. Bachrach, J. Beal, and J. McLurkin. Composable continuous-space programs for robotic swarms. *Neural Comput & Applic*, 19(6):825–847, 2010.

[28] J. . Baillie. Urbi: towards a universal robotic low-level programming language. In *International Conference on Intelligent Robots and Systems*, pages 820–825, 2005.

[29] N. Beckman and J. Aldrich. A programming model for failure-prone, collaborative robots. In *International Workshop on Software Development and Integration in Robotics*, 2007.

[30] I. Bekmezci, O. K. Sahingoz, and Ş. Temel. Flying ad-hoc networks (fanets): A survey. *Ad Hoc Networks*, 11(3):1254–1270, 2013.

[31] M. Bell. Service-oriented modeling. *John Willey & Sons*, 2008.

[32] J. Bellingham, M. Tillerson, A. Richards, and J. P. How. Multi-task allocation and path planning for cooperating uavs. In *Cooperative Control: Models, Applications and Algorithms*, pages 23–41. 2003.

[33] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[34] Y. Bokyung and K. H. Seok. A scheme improving performance of ieee 802.11 multicast protocol. In *1st International Conference on Communications, Computation, Networks and Technologies*. International Academy, Research, and Industry Association, 2012.

[35] N. Budhiraja and K. Marzullo. Tradeoffs in implementing primary-backup protocols. In *7th Symposium on Parallel and Distributed Processing*, pages 280–288, 1995.

[36] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In *Distributed Systems (2Nd Ed.)*. 1993.

[37] M. Chan, C. Chen, J. Huang, T. Kuo, L. Yen, and C. Tseng. Opennet: A simulator for software-defined wireless local area network. In *Wireless Communications and Networking Conference*, pages 3332–3336, 2014.

[38] A. Chandra and D. Mosberger. Scalability of linux event-dispatch mechanisms. In *USENIX Annual Technical Conference*, pages 231–244, 2001.

[39] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[40] H. Chao, Y. Cao, and Y. Chen. Autopilots for small unmanned aerial vehicles: A survey. *International Journal of Control, Automation and Systems*, 8(1):36–44, 2010.

[41] Y. Chen and X. Bai. On robotics applications in service-oriented architecture. In *The 28th International Conference on Distributed Computing Systems Workshops*, pages 551–556, 2008.

[42] Y. Chen, Z. Du, and M. Garca-Acosta. Robot as a service in cloud computing. In *5th International Symposium on Service Oriented System Engineering*, pages 151–158, 2010.

[43] S. Choi, N. Choi, Y. Seok, T. Kwon, and Y. Choi. Leader-based rate adaptive multicasting for wireless lans. In *GLOBECOM 2007 - Global Telecommunications Conference*, pages 3656–3660, 2007.

[44] E. Christopoulou and A. Kameas. Ontology-driven composition of service-oriented ubiquitous computing applications. In *3rd International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, pages 23–26, 2004.

[45] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2011.

[46] G. C. S. Cruz and P. M. M. Encarnação. Obstacle avoidance for unmanned aerial vehicles. *Journal of Intelligent & Robotic Systems*, 65(1-4):203–217, 2012.

[47] K. Dantu, B. Kate, J. Waterman, P. Bailis, and M. Welsh. Programming micro-aerial vehicle swarms with karma. In *9th Conference on Embedded Networked Sensor Systems*, 2011.

[48] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[49] A. El-Hoiydi and J.-D. Decotignie. Low power downlink mac protocols for infrastructure wireless sensor networks. *Mob. Netw. Appl.*, 10(5):675–690, Oct. 2005.

[50] S. Enderle, H. Utz, S. Sablatnög, S. Simon, G. Kraetzschmar, and G. Palm. Miro: Middleware for autonomous mobile robots. In *In Telematics Applications in Automation and Robotics*, 2001.

[51] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall, 2005.

[52] B. S. Faial, F. G. Costa, G. Pessin, J. Ueyama, H. Freitas, A. Colombo, P. H. Fini, L. Villas, F. S. Osrio, P. A. Vargas, and T. Braun. The use of unmanned aerial vehicles and wireless sensor networks for spraying pesticides. *Journal of Systems Architecture*, 60(4):393 – 404, 2014.

[53] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Virtual synchrony guarantees for cyber-physical systems. In *32nd International Symposium on Reliable Distributed Systems*, pages 20–30, 2013.

[54] M. J. Fischer, N. D. Griffeth, and N. A. Lynch. Global states of a distributed system. *Transactions on Software Engineering*, SE-8(3):198–202, 1982.

[55] M. Forte, W. L. de Souza, and A. F. do Prado. Using ontologies and web services for content adaptation in ubiquitous computing. *Journal of Systems and Software*, 81(3):368–381, 2008.

[56] J. Fuentes-Pacheco, J. Ruiz-Ascencio, and J. M. Rendón-Mancha. Visual simultaneous localization and mapping: a survey. *Artificial Intelligence Review*, 43(1):55–81, 2015.

[57] E. Galceran and M. Carreras. A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, 61(12):1258 – 1276, 2013.

[58] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Object-Oriented Programming*. 1993.

[59] B. P. Gerkey and M. J. Mataric. Pusher-watcher: an approach to fault-tolerant tightly-coupled robot coordination. In *International Conference on Robotics and Automation*, 2002.

[60] S. Griffiths, J. Saunders, A. Curtis, B. Barber, T. McLain, and R. Beard. Maximizing miniature aerial vehicles. *IEEE Robotics Automation Magazine*, 13(3):34–43, 2006.

[61] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies*, pages 38–57, 1996.

[62] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

[63] A. Y. Javaid, W. Sun, and M. Alam. UAVSim: A simulation testbed for unmanned aerial vehicle network cyber security analysis. In *IEEE Globecom Workshops*, pages 1432–1436, 2013.

[64] A. Kamerman and L. Monteban. WaveLAN®-II: a high-performance wireless LAN for the unlicensed band. *Bell Labs Tech. J.*, 2(3):118–133, 1997.

[65] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. Xors in the air: Practical wireless network coding. In *Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 243–254, 2006.

[66] S. M. Kay. *Fundamentals of Statistical Signal Processing, Volume I: Estimation Theory.* Prentice Hall, 1993.

[67] M. Koutsoubelias, A. Argyriou, and S. Lalis. Scalable and adaptive polling protocol for concurrent wireless sensor data flows. In *International Conference on Pervasive Computing and Communications Workshops*, 2018.

[68] M. Koutsoubelias, N. Grigoropoulos, and S. Lalis. Virtual Sensor Services for Simulated Mobile Nodes. In *Sensors Applications Symposium*, pages 1–6, 2017.

[69] M. Koutsoubelias, N. Grigoropoulos, and S. Lalis. A modular simulation environment for multiple uavs with virtual wifi and sensing capability. In *Sensors Applications Symposium*, pages 1–6, 2018.

[70] M. Koutsoubelias and S. Lalis. Coordinated broadcast-based request-reply and group management for tightly-coupled wireless systems. In *22nd International Conference on Parallel and Distributed Systems*, pages 1163–1168, 2016.

[71] M. Koutsoubelias and S. Lalis. Tecola: A programming framework for dynamic and heterogeneous robotic teams. In *13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 115–124, 2016.

[72] M. Koutsoubelias and S. Lalis. Fault-tolerance support for mobile robotic applications. In *13th International Symposium on Industrial Embedded Systems*, pages 1–10, 2018.

[73] J. Kuri and S. K. Kasera. Reliable multicast in multi-access wireless lans. In *Conference on Computer Communications. Proceedings.*, volume 2, pages 760–767, 1999.

[74] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *9th SIGCOMM Workshop on Hot Topics in Networks*, pages 19:1–19:6, 2010.

[75] C.-S. Lee, K.-H. Lee, and J.-K. Lee. A group rpc protocol for distributed systems. In *International Conference on Information, Communications and Signal Processing.*, volume 2, pages 805–809, 1997.

[76] K.-J. Lin and J. D. Gannon. Atomic remote procedure call. *Transactions on Software Engineering*, 11(10):1126–1135, 1985.

[77] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7):260–267, 1988.

[78] I. Lugo-Cardenas, S. Salazar, and R. Lozano. The MAV3dsim hardware in the loop simulation platform for research and validation of UAV controllers. In *International Conference on Unmanned Aircraft Systems*, pages 1335–1341, 2016.

[79] L. Mottola, M. Moretta, K. Whitehouse, and C. Ghezzi. Team-level programming of drone sensor networks. In *12th Conference on Embedded Network Sensor Systems*, pages 177–190, 2014.

[80] T. Murphy VII, K. Crary, and R. Harper. Distributed control flow with classical modal logic. In *Computer Science Logic*, pages 51–69, 2005.

[81] P. M. Newman. Moos-mission orientated operating suite. *Technical Report, Ocean Engineering Dept., Massachusetts Institute of Technology*, 2002.

[82] J. Paek and R. Govindan. Rcrt: Rate-controlled reliable transport protocol for wireless sensor networks. *ACM Trans. Sen. Netw.*, 7(3):1–45, 2010.

[83] L. E. Parker. Alliance: an architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240, 1998.

[84] O. Parodi, L. Lapierre, and B. Jouvencel. Hardware-in-the-loop simulators for multi-vehicles scenarios: survey on existing solutions and proposal of a new architecture. In *Proc. International Conference on Intelligent Robots and Systems*, pages 225–230, 2009.

[85] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from bell labs, 1995.

[86] C. Pinciroli and G. Beltrame. Buzz: An extensible programming language for heterogeneous swarm robotics. In *International Conference on Intelligent Robots and Systems*, pages 3794–3800, 2016.

[87] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[88] S. Rasmussen, J. Mitchell, C. Schulz, C. Schumacher, and P. Chandler. A multiple UAV simulation for researchers. In *Modeling and Simulation Technologies Conference and Exhibit*, pages 11–18, 2003.

[89] I. Rhee, A. Warrier, M. Aia, J. Min, and M. L. Sichitiu. Z-mac: A hybrid mac for wireless sensor networks. *IEEE/ACM Transactions on Networking*, 16:511–524, 2008.

[90] D. L. Russell. State restoration in systems of communicating processes. *Transactions on Software Engineering*, SE-6(2):183–194, 1980.

[91] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.

[92] R. C. B. Sampaio, M. Becker, A. A. G. Siqueira, L. W. Freschi, and M. P. Montanher. FVMS: A novel SiL approach on the evaluation of controllers for autonomous MAV. In *Aerospace Conference*, pages 1–8, 2013.

[93] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, Aug. 1985.

[94] M.-T. Sun, L. Huang, A. Arora, and T.-H. Lai. Reliable mac layer multicast in ieee 802.11 wireless networks. In *Proceedings International Conference on Parallel Processing*, pages 527–536, 2002.

[95] Y. Tamir and C. H. Sequin. Error recovery in multicomputers using global checkpoints. In *In 1984 International Conference on Parallel Processing*, pages 32–41, 1984.

[96] A. S. Tanenbaum. *Distributed Operating Systems*. 1995.

[97] K. Tang and M. Gerla. Random access MAC for efficient broadcast support in ad hoc networks. In *2000 IEEE Wireless Communications and Networking Conference. Conference Record (Cat. No.00TH8540)*. Institute of Electrical & Electronics Engineers (IEEE), 2000.

[98] K. Tang and M. Gerla. MAC reliable broadcast in ad hoc networks. In *2001 MIL-COM Proceedings Communications for Network-Centric Operations: Creating the Information Force (Cat. No.01CH37277)*. Institute of Electrical & Electronics Engineers (IEEE), 2001.

[99] P. Tripicchio, M. Satler, G. Dabisias, E. Ruffaldi, and C. A. Avizzano. Towards smart farming and sustainable agriculture with drones. In *International Conference on Intelligent Environments*, pages 140–143, 2015.

[100] T. van Dam and K. Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 171–180, 2003.

[101] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, pages 1–10, 2008.

[102] T. Vogeltanz and R. Jaek. Jsbsim library for flight dynamics modelling of a mini-uav. *AIP Conference Proceedings*, 1648(1):550015, 2015.

[103] X. Wang, A. Chowdhery, and M. Chiang. Networked drone cameras for sports streaming. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, volume 00, pages 308–318, 2017.

[104] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1567–1576, 2002.

[105] W. Ye, F. Silva, and J. Heidemann. Ultra-low duty cycle mac with scheduled channel polling. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 321–334, 2006.

[106] H. Zhang, A. Arora, Y.-r. Choi, and M. G. Gouda. Reliable bursty convergecast in wireless sensor networks. In *Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 266–276, 2005.

[107] X. Zhou, J. Guo, S. Durrani, and H. Yanikomeroglu. Uplink coverage performance of an underlay drone cell for temporary events. *CoRR*, abs/1801.05948, 2018.