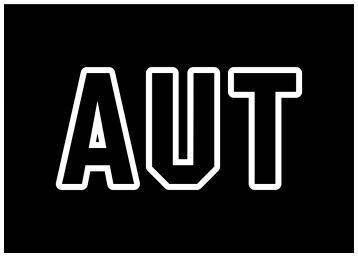

Design-for-Failure in Multi-Robot Systems through Integrated Architectures and Proactive Successor Allocation

Dong Hu

Supervisor: Dr Jing (Julia) Ma

School of Engineering, Computer & Mathematical Sciences
Auckland University of Technology

A thesis submitted to Auckland University of Technology
in fulfilment of the requirements for the degree of
Master of Computer and Information Sciences

May 2026

To my loving parents.

Copyright

Declaration

I hereby declare that, except where specific reference is made to the work of Others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgments.

Dong Hu
May 2026

Artificial Intelligence Declaration

This declaration follows the AUT guidelines regarding the use of AI tools in research.

Chapter number(s)	All Chapters (Language Polishing); Chapters 4 & 5 (Code Debugging & LaTeX Tables)
Purpose of AI Use	1. Spelling and grammar checking, and academic tone enhancement using Grammarly. 2. Debugging errors in the Python experimental code. 3. Optimizing LaTeX table formatting in Overleaf.
AI Tool(s) Used	Grammarly, Gemini
Prompts Used	"Why did our code have such an error?" "Provide a LaTeX template for a multi-column comparison table."
Output Received	Refined text suggestions, corrected code errors, and LaTeX formatting scripts.
Post-AI Processing Methods	All AI-generated suggestions were manually reviewed and verified. For text, I ensured the original research intent was maintained. For code and LaTeX, I conducted manual testing in Colab and adjusted formatting to meet thesis standards.

Free text box (Optional explanation of intent):

The use of AI tools was strictly auxiliary, focusing on linguistic precision and technical presentation. All core intellectual contributions, specifically the SPA-CBPA protocol design and data analysis, are entirely my original work.

Student Name: Dong Hu **Signature:**

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Jing (Julia) Ma, for her invaluable guidance, professional insights, and continuous encouragement throughout the entire research process. Her expertise in multi-robot systems and fault-tolerant technologies forms the foundation of this thesis.

At the same time, I would like to express my gratitude to all the faculty members of the School of Engineering, Computer Science, and Mathematical Sciences at the Auckland University of Technology for providing the necessary academic environment and resource support for my academic research. Finally, I would like to dedicate this research result to my loving parents. It is your unwavering support and selfless dedication that have enabled me to complete my academic journey.

Abstract

Modern warehouse multi-robot systems (MRS) are required to sustain high operational efficiency while remaining resilient to inevitable robot failures. However, traditional multi-robot task allocation (MRTA) research typically assumes fault-free environments. This assumption often results in centralized recovery bottlenecks or excessive computational overhead when failures occur. To address these limitations, this thesis introduces a novel binary analysis framework that categorizes fault-tolerant architectures into two types: native and integrated. Native architectures embed resilience directly into the task allocation design. In contrast, integrated architectures achieve fault tolerance through modular coupling with recovery mechanisms.

A high-fidelity 2D simulation platform was developed to perform benchmark evaluations of four representative algorithms: the centralized native algorithm MRPF, the distributed native algorithm BFTC, and two integrated methods: FT-CBPA and FT-ACO + BA. Experimental results under different load conditions demonstrated that integrated fault-tolerant architectures, which combine high-performance allocation strategies with modular recovery mechanisms, consistently outperform native architectures that tightly embed recovery logic within the allocation process. Comparative simulations further reveal that integrated architectures achieve superior scalability and higher task completion rates under failure scenarios.

Building upon the advantages identified in the integrated approach, this thesis proposes SPA-CBPA (Successor Pre-Allocation Consensus-Based Payload Allocation). This method transforms fault recovery from a reactive, post-event reallocation process into a deterministic pre-event activation mechanism through proactive successor assignment. Consequently, it successfully reduces the computational complexity of fault recovery from $O(N)$ to almost $O(1)$. Extensive benchmarking indicates that, compared to reactive benchmarks, SPA-CBPA reduces the delay in fault recovery to nearly zero and decreases the overhead of recovery-related communication by up to 65.2%. Although instantaneous recovery results in a modest 20% to 38% increase in total makespan, the proposed method significantly improves predictability and operational reliability in mission-critical logistics environments.

Overall, this research concludes a “design-for-failure" paradigm, demonstrating that proactive successor pre-allocation is essential for developing the next generation of scalable, efficient, and resilient multi-robot systems.

Publications

Dong Hu & Jing Ma. A Proactive Fault-Recovery Framework for Multi-Robot Task Allocation Using Successor Pre-Assignment. Accepted in 2026 29th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2026)

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Research Background

With the global advancement of Industry 4.0, multi-robot systems (MRS) have become foundational infrastructures in robotic mobile fulfillment systems [1], modern warehousing, and automated production lines. Multi-robot systems, through their inherent parallelism, scalability, and distributed robustness, can operate efficiently in complex and dynamic environments, significantly enhancing productivity, reducing operational costs, and improving service-level agreements (SLA).

Within these complex operational contexts, the Multi-Robot Task Allocation (MRTA) problem plays a crucial role in ensuring overall system efficiency. According to formal taxonomies [2], the core objective of MRTA is to assign a set of tasks to a team of robots to optimize global performance metrics, such as minimizing makespan, while satisfying strict operational constraints.

However, traditional MRTA research has predominantly emphasized efficiency optimality under idealized and fault-free assumptions. In real-world deployments, disturbances such as robot crashes, communication instability, sensor failures, and dynamic task changes are not exceptional events but rather normal operating conditions. When a robot failure is not properly addressed, the impact extends beyond the loss of a single task. It may disrupt task dependencies and trigger cascading failures at the system-level [3]. In mission-critical logistics environments, the failure of a single robot can halt an entire sorting line, leading to significant downtime and economic losses. Consequently, fault tolerance must be elevated from an auxiliary feature to a primary design principle.

This shift motivates a transition from pursuing a simple “optimal efficiency" to achieving “resilience efficiency" [4]. A modern multi-robot system not only needs to operate efficiently, but also recover rapidly and predictably from disruptions. Recovery latency, communication overhead, and task continuity become key performance indicators that directly influence SLA compliance and operational expenditure (OPEX). Designing task-allocation mechanisms that balance high performance with strong resilience is therefore a fundamental

challenge in scalable MRS deployment. Existing fault-tolerant approaches in MRTA are commonly categorized along dimensions such as reactive versus proactive recovery strategies, or centralized versus distributed control architectures. While these taxonomies are useful, we overlook a more fundamental architectural question: whether fault tolerance should be natively embedded within the task allocation core or integrated as a modular recovery mechanism.

This architectural decision significantly affects scalability, communication complexity, and recovery behavior. Crucially, this foundational choice fundamentally shapes the research background of this dissertation by shifting the analytical paradigm from localized algorithmic optimizations to structural blueprints at the system-level. Instead of treating fault tolerance as an isolated programmatic patch, this decision reframes the entire research context around the intrinsic trade-offs predetermined at the design stage. Consequently, it establishes the structural scaffolding that shapes the systematic taxonomy of literature in Chapter 2, guides the formal paradigm formulations in Chapter 3, and redefines the empirical benchmark coordinates for resilient multi-robot scheduling in high-throughput logistics.

Native fault-tolerant designs often introduce substantial coordination overhead to handle extreme failure scenarios, which can affect performance under normal conditions. Conversely, high-performance allocation algorithms achieve strong nominal efficiency but typically lack built-in fault tolerance and rely on external recovery protocols—such as passive re-auctioning—which may incur considerable recovery delays. This dissertation addresses this architectural gap through a structured analytical and experimental investigation.

The critical architectural gap identified in this research is the lack of a systematic and quantitative comparison between these two paradigms under unified experimental conditions and standardized evaluation metrics. In the existing literature, native and integrated strategies are treated as isolated solutions, obscuring the true trade-offs regarding who initiates recovery, how much coordination overhead is triggered, and what precise latency is incurred at the system level. Consequently, bridging this architectural gap directly drives the formulation of our three core research questions (RQs).

The inherent tension between baseline allocation efficiency and tightly coupled fault-tolerant overhead leads directly to RQ1 (Architecture), which demands a rigorous comparative baseline between native cores and decoupled integrated architectures.

The necessity to identify the operational boundaries and scalability of these paradigms across varying workload ratios motivates RQ2 (Trade-offs).

Finally, the systemic latency bottlenecks identified in current integrated baselines (i.e., the communication surge of reactive re-auctioning) provide the immediate catalyst for RQ3 (Proactive recovery), exploring whether a proactive successor pre-allocation overlay can achieve deterministic, near-zero latency recovery without compromising throughput.

1.2 Problem Statement and Research Motivation

Although performance-oriented MRTA algorithms have been extensively optimized in idealized settings, relatively little attention has been paid to how different architectural designs behave under realistic failures and communication variability. Existing work either embeds robustness directly within the allocation core or applies system-level recovery, such as re-auction after failure. Few studies systematically compare these strategies under unified experimental conditions and evaluation metrics. This lack of architectural-level comparison obscures the true cost of recovery, especially who initiates recovery, how much coordination is triggered, and what latency and communication overhead are incurred.

1.3 Research Questions and Objectives

To address the structural gaps identified in the multi-robot task allocation (MRTA) literature, this dissertation formulates its core scientific inquiry around three Research Questions (RQs). To ensure engineering clarity throughout this investigation, we establish the following formal definitions: *failure* is defined as a permanent fail-stop node crash or permanent communication network disconnection; *recovery latency* represents the elapsed physical time between the initial fault detection and the final successor takeover or re-auction assignment confirmation; and *communication overhead* is quantified as the total count of protocol messages exchanged explicitly to achieve fault mitigation.

1.3.1 Research Questions

RQ1 (Architecture). Under realistic failures and communication variability, how do native fault-tolerant cores compare with integrated architectures that decouple performance-oriented allocation from lightweight recovery modules?
Background: Current resilient systems often couple allocation with real-time fault mitigation, leading to over-engineered control loops. A rigorous comparison under unified baselines is essential to reveal the fundamental trade-offs between embedded resilience and modular integration under network perturbations.

RQ2 (Trade-offs). Which baselines better preserve task completion rate and makespan while bounding recovery latency and coordination overhead across task-to-robot ratios?
Background: System throughput often deteriorates unpredictably when a robot suffers a terminal hardware failure, making it vital to map this performance degradation across operational pressures. Understanding how completion rates and makespan scale with workload ratios allows designers to systematically identify the exact scalability boundaries of opposing paradigms.

RQ3 (Proactive recovery). By leveraging proactive-reactive detection strategies [5], can successor-based recovery eliminate system-wide re-auctioning for common single-robot faults without sacrificing throughput?
Background: Standard reactive recovery protocols rely on intensive post-event re-auction surges that stall healthy robots and trigger severe communication bottlenecks. Transitioning toward an architecture-driven proactive pre-allocation model can internalize contingency plans during the nominal consensus phase, turning failure responses into instantaneous, localized routing events.

1.3.2 Research Objectives

To answer these research questions, this dissertation aims to conduct an in-depth exploration and optimization of the fault-tolerant recovery strategies for multi-robot systems. Three primary research objectives are as follows:

- **Establish a Binary Architectural Framework:** Propose a novel binary analytical framework of "Native Fault-Tolerant Architecture" and "Integrated Fault-Tolerant Architecture" based on the fundamental distinction between the inherent (By Design) and post-integration (By Integration) nature of fault-tolerance capabilities.
- **Conduct Unified Implementation and Benchmarking:** Conduct a comprehensive algorithm-level implementation and high-fidelity simulation for four representative algorithms (MRPF, BFTC, FT-CBPA, and FT-ACO + BA) to enable the first systematic, quantitative comparison under consistent evaluation metrics.
- **Design a Proactive Successor-Based Recovery Protocol (SPA-CBPA):** Formulate and validate an innovative active fault-tolerant recovery protocol that pre-specifies a "successor" during the consensus stage, transforming fault recovery from a resource-intensive re-auction to a near-zero-delay planned activation.

1.4 Innovation Contributions

These three interconnected contributions form the core of this dissertation's investigation into resilient multi-robot systems.

In the theoretical Contribution, This paper introduces a novel binary "native vs. integrated" fault-tolerance architecture analysis framework, reframing the resilience paradigm around trade-offs predetermined at the design stage and providing a clear structural framework for classifying fault-tolerant MRTA algorithms.

In the methodological Contribution, We propose the proactive SPA-CBPA recovery protocol, which internalizes emergency backup plans into the decentralized consensus process. Using architectural determinism, this method successfully reduces the computational com-

plexity of fault recovery from $O(N)$ to near $O(1)$, transforming catastrophic reallocation into a localized activation event.

In the experimental Contribution, Through high-fidelity benchmarking of four representative baseline algorithms under unified conditions, this study quantitatively demonstrates that the integrated architecture balances nominal performance and resilience. Furthermore, empirical results verify that SPA-CBPA reduces fault recovery latency to nearly zero and reduces recovery-related communication overhead by up to 65.2%, despite a modest 20% to 38% increase in total makespan.

1.5 The Structure of Dissertation

The organizational structure of this dissertation follows the aforementioned research logic, and the specific arrangement is as follows.

Chapter 2 reviews related literature on multi-robot task allocation and fault-tolerant strategies, providing a theoretical foundation for the new framework and algorithms proposed in this study.

Chapter 3 presents the proposed native vs. integrated analytical framework and provides a theoretical comparison of existing representative fault-tolerant strategies.

Chapter 4 introduces the experimental setup and conducts a comprehensive benchmarking to identify the strongest integrated baseline (FT-CBPA).

Chapter 5 presents the SPA-CBPA framework and validates its advantages in recovery latency and communication overhead.

Chapter 6 summarizes and concludes the entire dissertation and describes future research directions.

Chapter 2

Literature Review

This chapter follows a funnel-structured narrative approach, beginning with a broad overview and progressively narrowing its focus. It starts with the general field of multi-robot task allocation and then successively examines fault-tolerant system architectures and the evolution of fault recovery mechanisms. The chapter is organized as follows: Section 2.1 introduces the fundamental definitions of the multi-robot task allocation (MRTA) problem, along with its mainstream classification frameworks and solution paradigms. Section 2.2 explains why “resilience" has become a key research topic beyond traditional “optimality" and categorizes existing strategies by recovery timing (Proactive vs. Reactive) and control architecture. Section 2.3 reviews two fundamentally different fault-tolerant design philosophies: “native fault-tolerant architecture" and “integrated fault-tolerant architecture," revealing the fundamental trade-offs between them. Finally, Section 2.4 synthesizes the aforementioned content, precisely outlines the existing gaps in current research, and clarifies the research direction of this study.

2.1 Review of Multi-Robot Task Allocation (MRTA)

2.1.1 Formal Definition and Classification of MRTA Issues

Multi-Robot Systems (MRS) have demonstrated significant potential in various fields such as automated warehousing and logistics distribution due to their advantages in handling complex tasks, enhancing performance, and improving reliability. In these systems, a key challenge is to optimally allocate a set of tasks among a group of robots, which is formally defined as the “Multi-Robot Task Allocation" (MRTA) problem [2]. This problem typically aims to find an optimal task allocation scheme to optimize a certain global performance metric.

To systematically investigate the MRTA problem, Gerkey and Matarić [2] proposed a widely adopted classification framework that characterizes the problem from three dimensions: whether the robots are performing a single task (ST) or multiple tasks (Multi-Task, MT); whether the tasks require a single robot (SR) or multiple robots (Multi-Robot, MR); and

whether the task allocation is Instantaneous Assignment (IA) or continuous Time-extended Assignment (TA) [2]. Although this classification method has a profound impact, as research progresses, scholars have found that it is difficult to cover increasingly complex task dependencies and constraints. Therefore, Korsah [6] proposed a more comprehensive classification method, which further considers the coupling relationships between tasks (such as time or spatial constraints), providing a more detailed description framework for more realistic MRTA problems. The research in this paper focuses mainly on the ST-SR-TA category, which is closely aligned with the automated warehouse application scenarios [7].

Although the classification method laid a solid foundation for the formal definition of MRTA problems, most of these early studies focused on idealized static environments, with their core goal being to pursue “efficiency optimality" (such as minimizing total cost or completion time) [8]. However, in real physical deployments like automated warehouses, the systems are essentially highly dynamic and full of uncertainties. Multiple systematic reviews have shown that MRTA research has gradually expanded from the early small-scale, static, and ideal communication environments to dynamic tasks, heterogeneous robots, and uncertain environments. The latest systematic review by Athira et al. in ACM Computing Surveys [9] emphasizes that in real applications, uncertainty and task perturbations are the norm. Algorithms not only need to pursue cost optimality but also need to take into account computational efficiency and operational robustness.

Therefore, some research in the academic field has shifted from the simple concept of “optimal efficiency" to “resilient efficiency". In this context, it is crucial to precisely distinguish between “robustness" and "resilience", according to the key classification method proposed by Prorok et al [10].

Robustness: Generally, it refers to the ability of a system to maintain its performance level when confronted with disturbances or model uncertainties (that is, the ability to resist interference).

Resilience: This concept focuses more on the system’s ability to recover its core functions and return to an acceptable performance level after encountering major failures (such as permanent failure of robot nodes) that result in a significant decline in performance.

2.1.2 Mainstream Solution Paradigms of MRTA

The current MRTA solutions can be roughly classified into three categories [11] as follows:

- **Optimize model-driven methods**: These include integer programming, vehicle routing problem (VRP) variants, etc., which can provide global optimal or bounded suboptimal solutions, but have limited adaptability to large-scale online problems and frequent replanning, and are mostly used in centralized control or offline planning benchmarks.
- **Behavior and incentive-driven approach**: Represented by Parker’s Alliance, it achieves distributed coordination through internal motivation, inhibition mechanisms, and behavioral sets, and inherently possesses certain fault-tolerance and self-adaptive capabilities.

These methods have good robustness, but it is difficult to strictly control the overall performance, and the parameter adjustment cost is relatively high in regularized storage scenarios.

- **Market/Consensus-driven Approach**: Centered around the Consensus-Based Bundle Algorithm (CBBA) proposed by Choi et al., and its asynchronous version, coupled with constraint coupling extensions, the distributed allocation is accomplished through iterative "bidding + consensus", balancing scalability and theoretical properties. Recent works, such as Payload-aware CBP, introduce factors of payload and capability changes on this framework, making it more closely aligned with engineering applications.

2.2 From Optimality to Resilience

The field of Dependable Computing has provided a strict definition for "failure - error - degradation", and has incorporated reliability, availability, security, etc. into a unified "dependability" framework [3]. The pioneering work of Avizienis et al. laid the conceptual foundation for subsequent research on fault tolerance and security. Resilience Engineering further proposes that the system should possess four capabilities: "response, monitoring, anticipation, and learning", in order to maintain acceptable performance under expected and unexpected disturbances [12].

Based on the recent MRTA review, it can be observed that the research focus is shifting from a single "optimality" to "resilient efficiency": it is necessary to maintain performance close to the optimal level under normal conditions, while also ensuring the completion rate of tasks, limiting recovery delay and communication overhead, and avoiding "cascade collapse" in the event of robot failure and environmental disturbances [13].

To achieve flexibility, the academic community has classified and studied the design of Fault-Tolerant Systems (FTS) from multiple dimensions [14]. Before delving into specific algorithm architectures, it is necessary to first organize these mainstream classification perspectives. The existing fault-tolerant classification methods mainly focus on describing the operational characteristics of the system or targeting specific objectives:

- **Classification by Recovery Timing**: This is one of the most extensive classification methods. Zhang and Jiang [15] in their influential review classified fault-tolerant control (FTC) into two methods: "passive" and "active". The passive method takes faults into account during the design stage and uses a fixed controller to tolerate faults, while the active method relies on real-time fault detection and diagnosis (FDI) to reconfigure the control law (i.e., reconfigure) to deal with faults. This "active" approach is the theoretical basis for the "Proactive" (active) and "Reactive" (reactive) recovery mechanisms to be discussed in this section [16].

- **Classification by Control Architecture**: The implementation method of the fault-tolerant mechanism is closely related to the underlying control architecture. Khalastchi and Kalech

in 2019 [17], in their review on multi-robot fault diagnosis, classified the diagnostic methods into two categories: “centralized" and "distributed". The centralized system relies on a single decision node and is easy to coordinate, but also has the risk of a single point of failure; while the distributed system disperses the decision-making to each robot, which improves the robustness and scalability of the system.

- **Classification by Fault Type Handled**: Different systems are designed to handle different types of faults. For instance, some systems focus on handling “fail-stop" faults (where the robot stops operating after a failure), while others are designed to deal with more severe “Byzantine Faults", where faulty nodes may send arbitrary or even malicious error messages.

Although these classification methods (based on timing, architecture, and type) are crucial for understanding the properties of fault-tolerant systems, they fail to address a more fundamental issue: whether the fault-tolerance capability itself was “natively built-in" as a core function at the beginning of the algorithm design, or was "subsequently integrated" as an external module?

Therefore, this dissertation proposes a novel binary analysis framework, which classifies the existing work from the perspective of the construction paradigm of the algorithm, namely: Native Fault-Tolerant Architecture: “Designed for Fault Tolerance" and Integrated Fault-Tolerant Architecture: “Performance-Oriented Modularization". This framework will serve as the main thread for analyzing and comparing various algorithms in the subsequent chapters.

2.3 Review of Fault-Tolerant System Design

2.3.1 Native Fault-tolerant Architecture: “Design for Fault Tolerance" Approach

The native fault-tolerant architecture aims to provide deterministic reliability guarantees by embedding the fault-tolerance mechanism into the core of the algorithm [18]. Centralized implementations, such as the multi-robot preemptive task scheduling and fault recovery algorithm (MRPF) proposed byKalempa [19] [20], maintain the global state and make all decisions, including fault recovery, through a central controller. Although this architecture is responsive, the central controller constitutes a single point of failure and a performance bottleneck of the system, limiting its application in large-scale systems [21]. Distributed implementations aim to overcome the drawbacks of the centralized architecture. The most representative one is the system that handles Byzantine faults. Byzantine faults are the most challenging fault types in distributed systems, where nodes may exhibit arbitrary or even malicious behavior [22]. For example, the Byzantine Fault-Tolerant Consensus Algorithm (BFTC) proposed by Strawn and Ayanian [23] ensures the correctness and consistency of decisions by introducing a blockchain consensus protocol. However, this strong consistency

guarantee comes at the cost of high communication overhead and significant decision delay, making it difficult to adapt to high-throughput application scenarios.

2.3.2 Integrated Fault-tolerant Architecture: Modular “Performance-First" Approach

Unlike the native architecture, the integrated fault-tolerant architecture offers a more flexible design approach [24]. It enables researchers to first select a scheduling algorithm with the best performance in a fault-free environment as the core, and then add a modular fault-tolerant protocol to it [10]. The Alliance architecture proposed by Parker [25] is an early example of this idea. It is a behavior-based distributed framework that achieves dynamic task reallocation through internal motivations (such as “impatience" and “consent"), demonstrating the potential of modular fault-tolerance.

The Consensus Bundling Algorithm (CBBA) is one of the ideal foundations of modern integrated architectures. As a pioneering distributed, market-based MRTA protocol, CBBA was proposed by Choi [26], and its core is a two-stage iterative process: Bundle Building and Consensus. Robots reach an agreement on task ownership through local communication, thereby forming an efficient allocation plan while resolving conflicts. CBBA is highly praised for its excellent scalability, robustness, and the ability to provide theoretically guaranteed, approximately optimal solutions. However, the original design of CBBA did not include mechanisms for handling robot physical failures, which makes it an ideal starting point for building integrated fault-tolerant architectures.

The success of CBBA has led to a wealth of subsequent research, such as Asynchronous CBBA (ACBBA) and Coupled Constraint CBBA (CCBBA). Robots only bid when they confirm that their load is sufficient to complete the task, making CBPA a more practical scheduling core.

2.4 Comprehensive Review and Research Gaps

The MRTA foundation is relatively mature, but it lacks a systematic “resilience assessment" perspective [27]. Although the existing reviews do cover uncertainty and robustness, there is still a lack of work that quantitatively compares different fault-tolerant architectures and recovery mechanisms for warehouse-type MRTA within a unified experimental framework. While recent scalable coordination methods address macro-level stability under network constraints, and advanced scheduling frameworks explore dynamic task reallocation primarily under soft execution delays, they rarely provide a quantitative architectural comparison under terminal node failures.

The native fault-tolerant architecture provides strong guarantees, but it is difficult to meet the efficiency and deployment cost requirements of large-scale storage systems. Methods such as MRPF and BFTC have verified the feasibility of “constructing fault tolerance within

the algorithm", but they have also exposed problems such as central bottlenecks and if communication complexity. The integrated architecture demonstrates great potential, but the design space has not yet been systematically explored. The “performance-first" cores, such as CBBA/CBPA, provide a platform for stacking multiple recovery mechanisms, but currently, most related work consists of scattered solutions and lacks a head-to-head comparison with the native architecture in a unified scenario.

The current recovery mechanisms mainly rely on reactive re-auctioning, with insufficient attention paid to proactive/preparatory mechanisms [28]. The re-auctioning-based solutions are simple to implement, but they incur excessive recovery costs under high load and frequent failure conditions; the preparatory mechanisms inspired by resilience engineering and emergency planning have not yet formed a lightweight and reproducible paradigm in consensus-based MRTA.

Based on the above observations, this dissertation focuses on two core issues: (i) conducting systematic reproduction and comparison of representative native and integrated fault-tolerant MRTA algorithms under the same simulation environment and evaluation indicators; (ii) designing and verifying a successor pre-allocation-driven SPA-CBPA proactive recovery protocol on top of the integrated CBPA kernel, to introduce the "emergency plan/preparatory proactive recovery" concept into multi-robot task allocation in an engineering-achievable manner. To clearly synthesize these structural distinctions, a taxonomic comparison of existing fault-tolerant paradigms is summarized in Table 2.1.

Table 2.1 Taxonomic Comparison of Fault-Tolerant MRTA Paradigms

Paradigm / Approach	**Control Structure**	**Recovery Timing**	**Fault Type Handled**	**Primary Limitations**
Centralized Native (MRPF)	Centralized	Reactive	Simple Physical	Single point of failure; low scaling
Distributed Native (BFTC)	Decentralized	Reactive	Byzantine Failures	Heavy consensus delay; low throughput
Heuristic Integrated	Decentralized	Reactive	Simple Physical	Weak random task addition; high delay
Field-Based Integrated	Decentralized	Reactive	Simple Physical	Post-event latency; message overhead

Chapter 3

Fault-Tolerant Scheduling Strategies in Warehouse Robotics

In modern intelligent warehouse systems, automated mobile robots (AMR) play a central role in handling, sorting, and transporting goods. However, the combination of complex workflows, densely populated operational environments, and dynamic uncertainties makes these systems highly vulnerable to failures and disruptions [17], such as actuator malfunctions, path blockages, communication losses, and task abandonment. Without effective failure management, these issues can not only degrade the performance of individual robots but also create systemic scheduling bottlenecks or even trigger complete operational breakdowns. Consequently, developing robust scheduling mechanisms with built-in fault tolerance has become a critical research area in warehouse robotics.

When classifying fault-tolerant mechanisms for multi-robot systems, there are various effective perspectives in the academic community. For instance, they can be categorized based on the recovery timing into “proactive" and “reactive" strategies [15], or classified according to the control structure into “centralized" and “distributed" systems [29]. Alternatively, they can be divided based on the specific types of faults handled, such as Byzantine faults [22] or fail-stop faults [30].

In multi-robot system design, a fundamental tension has long existed between maximizing performance and ensuring system robustness [10]. On one hand, the primary advantage of multi-robot systems lies in their high operational efficiency under fault-free conditions. On the other hand, real-world deployments are inherently uncertain, demanding strong fault-tolerance to handle hardware malfunctions, communication disruptions, and other unforeseen events. This intrinsic trade-off has led to the emergence of two fundamentally distinct paradigms for algorithm and system design.

- **Native Fault-Tolerant Architecture [31]**: Here, fault tolerance is embedded as an inherent element of the algorithm from the outset—Fault Tolerance by Design. The recovery logic is tightly integrated with the scheduling mechanism. For instance, the MRPF algorithm was designed with a complete fault recovery process from the beginning.

- **Integrated Fault-Tolerant Architecture**: In this case, a high-performance scheduling algorithm (whose original lack of built-in fault tolerance) acquires recovery capability through the addition or integration of an independent fault-tolerant module. For instance, the authors of the CBPA algorithm explicitly indicated that future work would involve extending fault-tolerance functionality, making it a typical case of the integration strategy.

This difference in design leads to the core research question of this chapter: When the objective is to maximize fault-tolerant recovery performance, does the inherent advantage of native architecture [32] outweigh that of integrating a carefully designed recovery module into an efficient scheduling core? To address this, this chapter conducts a critical analysis of four representative methods and highlights the fundamental trade-offs and intrinsic limitations of these two design paradigms in the context of warehousing and logistics systems.

Before detailing the specific algorithmic mechanics, a methodological clarification regarding the system-level diagrammatic representations is warranted. Although the runtime control logic, reactive decision loops, and error mitigation behaviors within the evaluated Multi-Robot Task Allocation (MRTA) paradigms inherently exhibit event-driven characteristics, flowcharts are intentionally and systematically retained for Figure 3.2, Figure 3.4, Figure 3.9, and Figure 3.10. This structural modeling choice is justified by the methodological imperative to maintain strict procedural consistency with the nominal allocation pipelines presented throughout this dissertation (e.g., the standard scheduling frameworks depicted in Figure 3.7 and Figure 3.8). By prioritizing the sequential execution flow, conditional deterministic branchings, and localized multi-agent message routing over discrete software operational states, these flowcharts provide a highly scannable, algorithm-centric abstraction. This approach more effectively elucidates the exact temporal ordering and programmatic stages of post-fault recovery protocols within a unified comparative benchmark, ensuring mathematical cohesion and narrative alignment across the text.

Table 3.1 Summary of High-Frequency Mathematical Symbols and Formal Definitions

Symbol	Formal Academic Definition and Operational Description
b_i	Ordered task bundling list maintained locally by robot i to track intended tasks.
p_i, π_i	Intended execution path or route routing sequence derived from the local bundle b_i.
c_{ij}, C_{ij}	Marginal insertion cost or utility score for robot i to accept candidate task j.
$\tilde{r}_j, r_j$	Multi-dimensional resource or physical payload requirements vector required by task j.
l_{ij}	Actual payload contribution or resource replenishment allocated by robot i to task j.
X	Global binary winner assignment matrix, where $x_{ijk} = 1$ denotes active task allocation.
Z	Proactive binary successor pre-allocation contingency matrix for emergency takeovers.
w_j	Primary designated winner robot assigned for executing and completing task j.
s_j	Pre-allocated emergency successor robot designated for instantaneous takeover of task j.
S_p	Total objective reward or systemic utility score associated with a specific execution path p.

3.1 Native Fault-Tolerant Architecture

3.1.1 Multi-Robot Preemptive Task Scheduling with Fault Recovery (MRPF)

MRPF [20] is a centralized fault-tolerant scheduling algorithm designed for task scheduling and execution management of Autonomous Mobile Robots (AMRs) in dynamic smart factory environments. It integrates priority-based preemptive scheduling with a multi-robot collaboration fault tolerance mechanism, enabling fast response and efficient recovery in an unstable environment.

Before examining the detailed mechanism of MRPF, it is important to first clarify the basic definition of task management for multi-robot systems as presented by the original author. The problem is decomposed into three levels [2]: task allocation (Assignment), which determines which robot or group of robots is responsible for a task; task scheduling (Scheduling), which defines the order of task execution; and task assignment (Allocation), which specifies when and how task instructions are dispatched to robots. MRPF functions as a unified scheduling framework that integrates these three levels. A key concept within this

framework is coalition formation—the dynamic creation of one or more robot teams tailored to the requirements of each task, rather than relying on rigid one-to-one allocations.

The MRPF system architecture is organized around a central controller that maintains a global task list (process_list). Each task is categorized into one of four priority levels: Critical, Major, Normal, and Minor, which dictate both the execution order and the number of robots assigned in Table 3.2. MRPF further integrates a fault recovery Mechanism based on dynamic substitution policies into the task scheduling architecture [19].

Table 3.2 Priority Classification of Task Types and Suggested Robot Allocation

Priority Level	Type	Description	Suggested Count
Priority 4	Critical	Emergency tasks (e.g., fire alarm, deadline delivery)	≥ 3
Priority 3	Major	High-impact tasks (e.g., bulk inbound/outbound)	≥ 2
Priority 2	Normal	Routine tasks	≥ 1
Priority 1	Minor	Non-essential tasks	Idle only

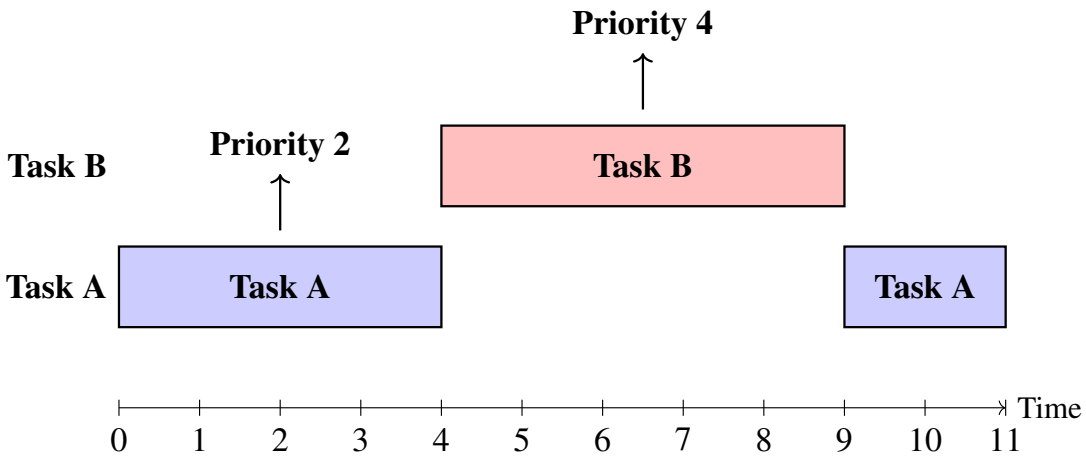

Fig. 3.1 Gantt-style task preemption diagram with priority annotations. Task A (Priority 2) is interrupted by Task B (Priority 4) and resumes execution afterward.

As shown in the Fig 3.1, the MRPF method allows high-priority tasks (such as Task B with a priority of 4) to preempt low-priority tasks (such as Task A with a priority of 2). However, as emphasized by the original author, this preemption does not occur arbitrarily. To maintain environmental stability and task integrity, preemption is only triggered once the robot completes its current basic operation, for example, finishing an "item retrieval from the shelf" or "delivering goods to the designated location." This ensures that tasks are not interrupted mid-process, such as during transportation, thereby preventing goods from being left in unsuitable positions within the warehouse.

During task execution, a robot may become unable to proceed due to low power, sensor failure, communication loss, or hardware failure. To prevent tasks from being completely disrupted, they should not be entirely interrupted. The MRPF mechanism suspends the robot's operations immediately upon fault detection and records its execution state in a structured

format. Once a fault is confirmed—either through the robot's diagnostic module or through scheduling feedback—the system initiates the following steps: First, the task's execution details (such as current phase, path progress, and cargo handling status) are cached in the task queue to support later recovery; Second, if the faulty robot is transporting goods, it is instructed to safely set them down to avoid potential damage or mishandling; At the same time, the fault unit is marked as undispatchable and directed to the Maintenance Sector. At this point, the system transitions into the task recovery preparation stage, which triggers the second phase: substitute robot selection and task resumption. In the second phase, MRPF applies a fault recovery mechanism based on clear and efficient principles. When a robot fails, the system identifies a replacement to continue the unfinished task. The original paper specifies straightforward selection criteria: the substitute must be the robot closest to the failure point and currently marked as available. If no idle robot is available, the scheduler may preempt a robot performing a low-priority task, thereby reallocating resources to ensure critical task completion. Once a substitute is chosen, it resumes the original task precisely from the point of interruption, avoiding the need to restart from the beginning and thereby enabling seamless task continuity. Finally, after being marked as "Unavailable", the fault robot is removed from the current scheduling graph and directed to the maintenance sector. In this state, it no longer participates in task scheduling until it is either restarted or manually repaired. Figure 3.2 shows the circular flowchart of the MRPF fault recovery mechanism.

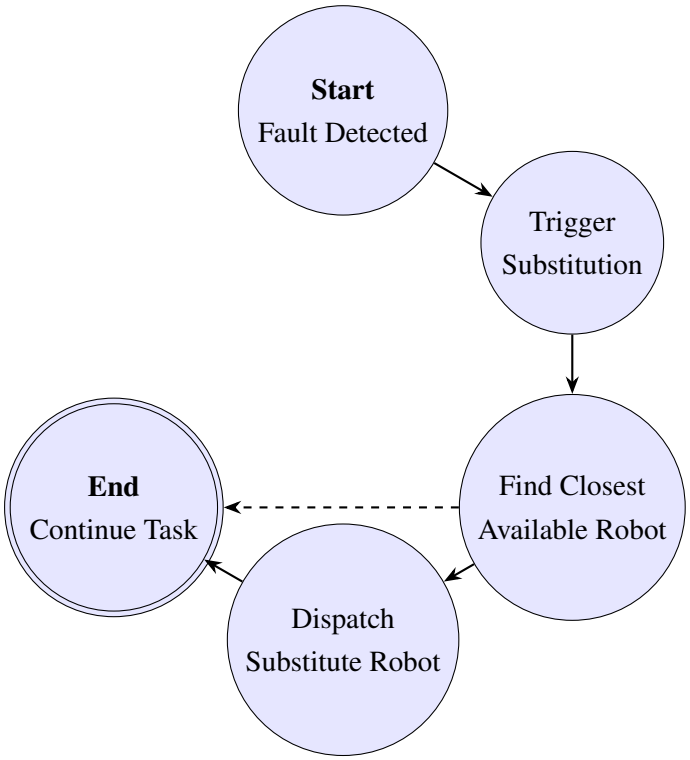

Fig. 3.2 Circular Flowchart of the MRPF Fault Recovery Mechanism

By analyzing the MRPF method and its performance in the original experiments, we can summarize its advantages and disadvantages in fault recovery as shown in the following table:

Table 3.3 Summary of Advantages and Limitations of MRPF Fault Recovery

Advantages	Limitations
✓ Seamless task recovery through context caching and reassignment.	✗ Recovery quality depends on the completeness and freshness of cached context. Incomplete data may delay or degrade restoration.
✓ Fast replacement via proximity-based selection.	✗ Excessive nesting of substitutions can increase system overhead and delay real-time performance.
✓ Recursive substitution: replacement robots can be re-substituted.	
✓ Priority-integrated scheduling for intelligent recovery handling.	✗ High-priority tasks may continuously preempt low-priority ones, causing starvation and fairness issues.

3.1.2 Byzantine Fault-Tolerant Consensus (BFTC)

Strawn and Ayanian [23] propose a blockchain-based fault-tolerant recovery mechanism to solve Byzantine Faults in multi-robot task assignment and path planning (MAPD) [23].

A Byzantine fault [22] represents one of the most challenging failure types in distributed systems. In such cases, certain nodes may behave unpredictably or even maliciously, leading to confusion and incorrect decisions within the rest of the system.

The BFTC method is implemented on the Tendermint blockchain platform, with a system architecture that consists of three core components: the robot agent, the Tendermint node network, and a customized application layer logic, namely the BFTC chain. The design emphasizes a clear separation between the Tendermint Core engine responsible for the network and consensus from the state machine that handles the specific application logic.

This separation is enabled through the application Blockchain Interface (ABCI), which acts as a bridge between the BFTC chain application and the underlying Tendermint consensus engine. ABCI defines two fundamentally different types of interaction messages as visualized in the preliminary assignment process shown in Figure 3.4:

- Query message: This operation retrieves the state from a node's local copy. It is a lightweight, read-only process that enables agents to quickly access global information, including the current timestep, available tasks, and planned paths.
- DeliverTX message: This operation is used to propose a state update to the entire network, such as submitting a new path or committing to a task. Since it requires triggering the BFT consensus protocol across all nodes, it is a costly write operation. Experimental results indicate that each DeliverTX call takes approximately 1–2 seconds to reach consensus.

With this architecture, the BFTC system preserves a distributed and tamper-proof global state ledger that primarily records timestep data, the task pool, allocated time-location pairs, and each robot's destination coordinates.

In the task allocation phase, BFTC utilized the characteristics of replicated state machines to achieve a novel form of decentralized global optimization. At the start of each new timestep, every non-byzantine node in the BFTC network runs the Hungarian algorithm [33] on its local copy of the global state. This algorithm determines the optimal task-to-robot pairing for the entire system. Once computed, the optimal allocation is recorded as part of the consensus state. As a result, robots do not need to engage in complex negotiations or bidding processes; instead, they simply issue a Query message to the blockchain, retrieve their assigned task, and proceed to execute it.

During the path planning stage, the robot applies the A* algorithm [34] to calculate a collision-free route using the usedElements data. If a task is assigned, it generates a complete path from its current position to the pick-up location and then to the delivery destination. If no task is assigned, the robot checks for potential collision risks. In the presence of risk, it conducts an exhaustive search to identify feasible safe endpoints and plans a path accordingly. If no conflict exists, it remains stationary. To reduce online computation load, the system supports A* with heuristic pre-computations (such as Manhattan distance).

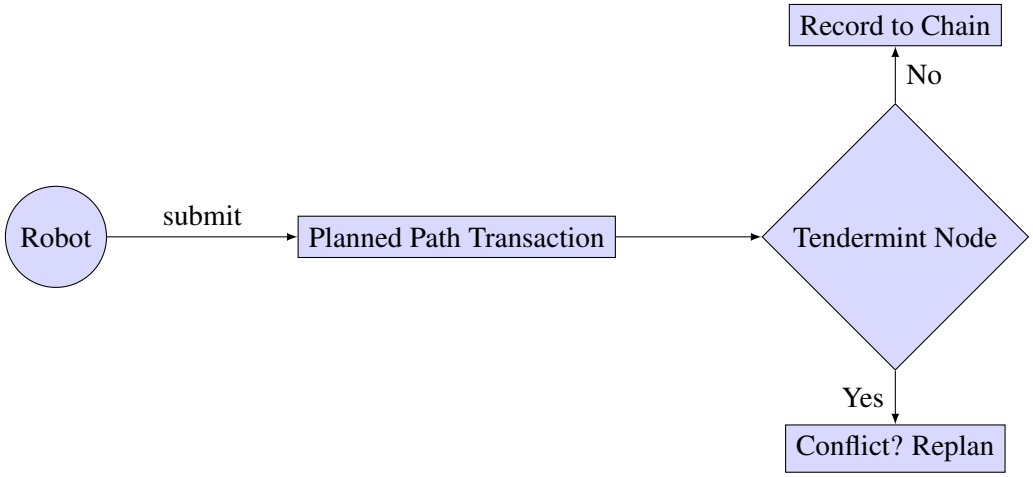

Fig. 3.3 Path verification process in BFTC

Once path planning is completed, the specific verification flow of these path-space-time pairs is illustrated in Figure 3.3. The robot packages the planned route into a DeliverTX transaction and submits it to the Tendermint blockchain network. This transaction includes detailed path-space-time pairs. All nodes participating in the consensus then verify its validity against the current on-chain state. If at least two-thirds of the nodes approve [23], the path is confirmed, recorded on the chain, and synchronized across all replicas. If validation fails, for example, there is a time-space conflict with existing paths or abnormal transaction data, the transaction is rejected, and the robot must re-plan and resubmit. Facilitating the rapid redistribution of orphan tasks as shown in the logical flow of Figure 3.4.

To guarantee that the entire distributed system progresses synchronously through discrete timesteps, BFTC introduces a special coordinating role, which is called 'Auxiliary Agent A'. This is not a physical robot but a logical system process. Its function is to wait until all robots have finished submitting their path planning transactions, after which it issues a special transaction to the blockchain to advance the system's timestep. During this process, it also incorporates any newly generated tasks into the task pool. By centralizing time advancement under this dedicated role, BFTC effectively avoids the deadlock issues commonly seen in traditional distributed systems during clock synchronization.

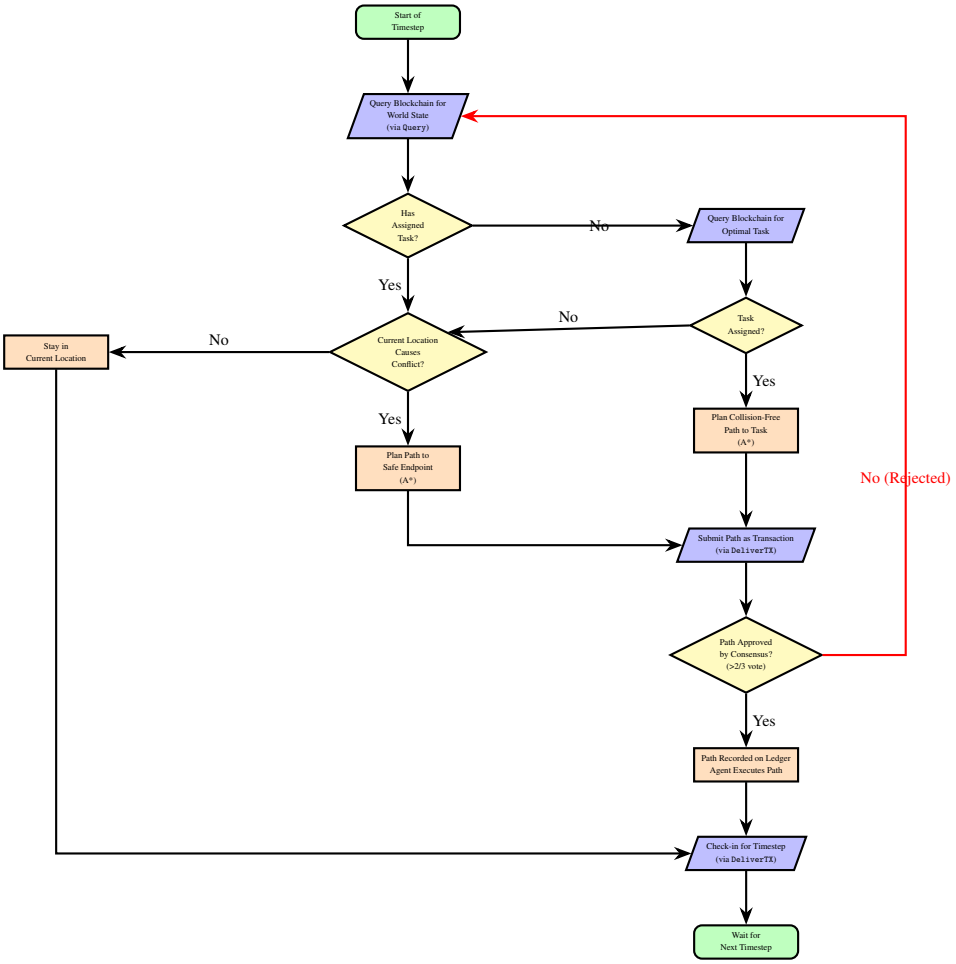

Fig. 3.4 Agent's Decision and Path Submission Cycle in BFTC

Table 3.4 Fault Recovery Process in the BFTC System

Step	Description
1	Robot queries the blockchain for the current state (timestep, used positions, endpoints, tasks).
2	If idle, plans a collision-free path using A*.
3	Submits the planned path to the blockchain as a transaction.
4	If approved by Tendermint consensus (more than 2/3 nodes), the robot executes the path.
5	If rejected (e.g., conflict), it replans automatically.

Table 3.5 Simplified Summary of Advantages and Limitations of BFTC

Advantages	Limitations
✓ Tolerates Byzantine agents and ensures correct task execution.	✗ Consensus introduces delays, affecting responsiveness.
✓ Performs comparably to other distributed MAPD methods.	✗ Lacks task swapping, limiting flexibility.
✓ Maintains moderate runtime growth as team size increases.	✗ Requires stable communication and computational capacity.

3.1.3 Summary of Native Fault-Tolerant Architecture Analysis

The analysis of two representative methods within the native fault-tolerant architecture, namely centralized MRPF and distributed BFTC, has been completed.

A comparison shows that MRPF, with its centralized global view, enables fast responses to simple physical faults but suffers from scalability bottlenecks and potentially suboptimal decision-making. In contrast, BFTC leverages distributed consensus to achieve strong protection against Byzantine faults, yet this comes at the cost of severe performance delays when aiming for high throughput in storage scenarios. These findings highlight an important principle. In the original architecture, the algorithm's top-level design largely dictates which types of faults the system can handle efficiently, along with its recovery performance characteristics. However, this rigid design-stage trade-off restricts the system's adaptability in complex and dynamic real-world environments.

This naturally leads us to ask: Is there a more flexible strategy that can decouple the high-performance scheduling core from the modular recovery protocol? This question provides both the motivation and the direction for our deeper exploration of an Integrated Fault-Tolerant Architecture.

3.2 Integrated Fault-Tolerant Architecture

3.2.1 Consensus-Based Payload Allocation (CBPA)

The Consensus-Based Bundle Algorithm (CBBA), introduced by Choi et al. in 2009 [26], is a distributed multi-robot task allocation method designed for environments with limited communication, dynamic changes, or strict scalability requirements. It addresses the challenge of enabling multiple agents to collaborate and complete tasks without conflicts. By combining the efficiency of market-based auctions with the robustness of consensus protocols, CBBA achieves global task allocation through local information exchange, eliminating the need for a central coordinator [35].

The algorithm operates in two phases: bundle construction and consensus. In the bundle construction phase, each agent builds its task execution sequence (or task bundle) based on local information. This step follows the principle of marginal gain maximization, where agents iteratively select the most cost-effective task and insert it into the optimal position within their planned path. Specifically, an agent initializes its task bundling list b_i and the execution path p_i. From the set of candidate tasks not yet locked by other agents, it calculates the marginal score $c_{ij}[b_i]$for each potential insertion. The marginal score is defined as the difference between the total path score after inserting a task and the current score of the path, that is:

$$c_{ij}[b_i] = \max_n \left(S_{p_i \oplus_n \{j\}} - S_{p_i} \right)$$

$\oplus_n\{j\}$ denotes inserting task j into the n-th position of path p_i, where S_p represents the total reward associated with path p.

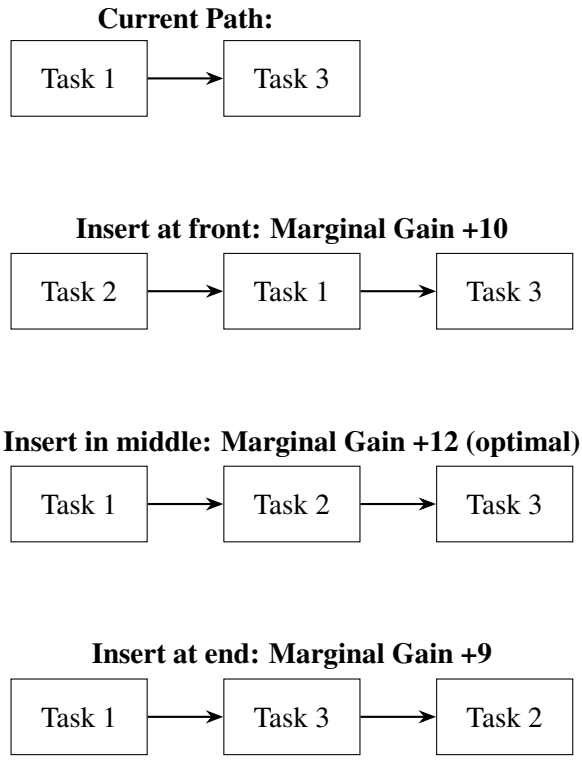

Fig. 3.5 Illustration of the task insertion process in CBBA. The agent evaluates different insertion points for Task 2 within its current path (Task 1 $\rightarrow$ Task 3), choosing the position with the highest marginal gain.

Figure 3.5 shows a typical example of task insertion evaluation for ease of understanding. Suppose the current path is [Task 1$\rightarrow$ Task 3], the candidate task to be evaluated is Task 2. Three different insertion positions are compared in the figure. Inserting Task 2 into the middle position can obtain the maximum gain. Therefore, the agent considers this to be the current optimal strategy. This process will continue until the task path constructed by the agent reaches the preset upper limit of task capacity, or there are currently no tasks that can bring positive marginal benefits.

The Consensus Phase is the second stage of the CBBA algorithm and serves as the foundation of its fault-tolerant recovery mechanism. In this stage, each agent broadcasts the local results from the bundling construction phase, including the task bid (i.e., marginal score), the Winning Agents List, the Winning Bids Matrix, and the information timestamp, to the communication neighbors. Upon receiving the message, agents compare the incoming data with their own local records. If they find that a certain task has been awarded by others with a higher score (including a conflict), a fault-tolerant rollback is triggered: the agent removes the conflicting task along with all subsequent tasks in its bundle, then readjusts its path and local state to preserve the global constraint that each task can only be assigned once. This process is shown in Figure 3.6, where both R1 and R2 attempt to claim Task 4. Since R2 places the higher bid, it wins the task. After receiving the consensus update, R1 detects the conflict, removes Task 4 along with any dependent tasks, and reverts to a conflict-free state.

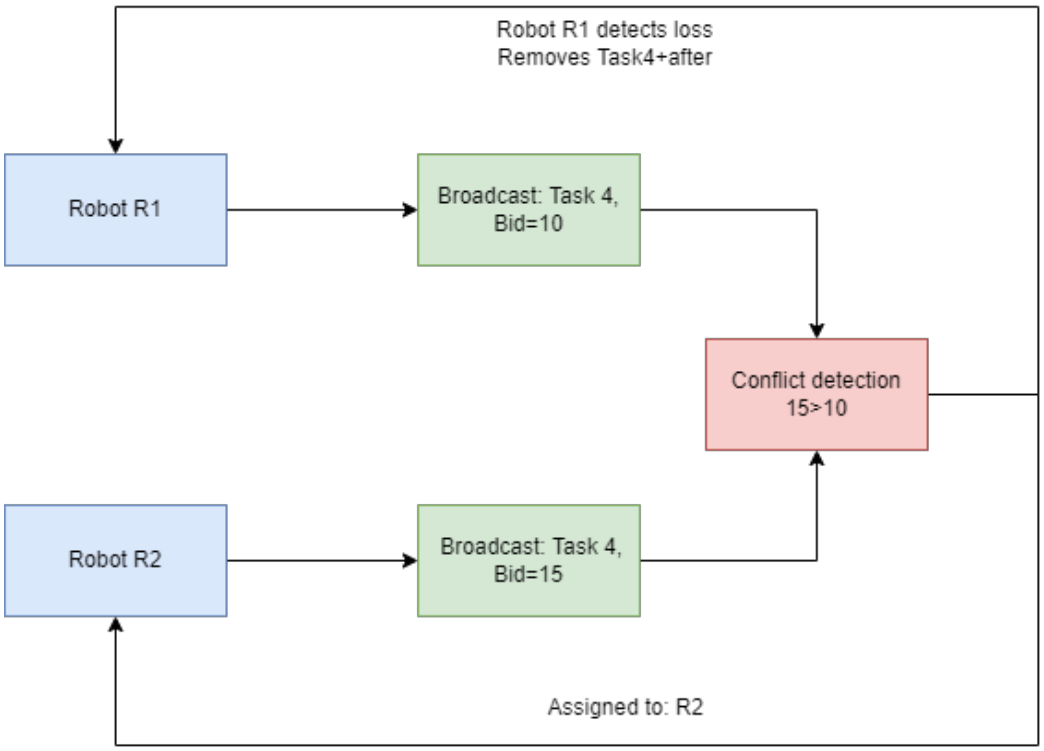

Fig. 3.6 Illustration of the consensus phase in CBBA. Robot R1 and R2 both bid for Task 4. Since R2 offers a higher bid, it wins the task. R1 detects the conflict and performs a fault-tolerant rollback, removing Task 4 and all subsequent tasks from its local bundle.

Although CBBA performs well in many scenarios, its standard framework does not explicitly address a key practical challenge in the task execution process: the management of the resources that the robot can consume (or referred to as “payload", such as energy, ammunition, duration of sensor usage, etc.). In traditional CBBA, the scoring function of the task mainly focuses on indicators like path length and execution time, while ignoring whether the robot’s payload is sufficient to complete the task. This may lead to the robot generating an apparently optimal task sequence during the planning stage, but failing to complete the task in actual execution due to resource depletion, resulting in task failure.

To address the challenges in task allocation based on load perception, Qiu et al. [36] extended the CBBA and proposed the Consensus-Based Payload Allocation (CBPA) algorithm. The core innovation of CBPA does not lie in changing the consensus mechanism, but rather in introducing explicit modeling and tracking of the robot’s Payload during the bundling construction stage. Its key contribution lies in introducing the effective load allocation matrix. CBPA uses a load allocation matrix to real-time track the resource inventory of each robot and the resource requirements of each task. It has improved the scoring function: CBPA modified the task scoring function and regarded resource feasibility as the core constraint for task selection. As shown in Equation 3.1, the new scoring function takes into account multiple factors:

$$S_i(p_i) = \sum_{j=1}^{N_t} \left[(\alpha \cdot m_{ij}^a + \beta \cdot m_{ij}^b + t_{ij}) \cdot x_{ijk} \right] \tag{3.1}$$

- m_{ij}^a, m_{ij}^b: indicate the payload compatibility between task j and robot i, such as whether robot i is capable of performing strike or reconnaissance tasks;
- t_{ij}: denotes the time cost of executing task j by robot i;
- α, β: weight coefficients that control the priority of different task types in the utility function;
- x_{ijk}: a binary indicator denoting whether task j is assigned to the k-th position in robot i's execution path.

This formulation enables robots to prioritize tasks not only based on marginal utility but also on their resource feasibility, allowing them to construct and recover execution paths more robustly and efficiently.

First, for each task j to be evaluated in the path, the system calculates whether the current remaining resources l_i of the robot i can meet the resource requirements $\tilde{r}_j$ of the task. If $l_i < \tilde{r}_j$, that is, the robot resources are insufficient to complete the task, then the task is regarded as “unexecutable" and needs to be rolled back from the path.

Second, the system further assesses whether the scheduling benefits are sufficient if the task resources are feasible. By calculating the score increment brought by inserting the task j into the n th position in the path:

$$\Delta S = S_{p_i \oplus_n j} - S_{p_i}$$

If the increment is negative, that is, the overall score of the path is reduced after the task is inserted, it indicates that the task brings negative returns and should be deleted first.

Table 3.6 Summary of Advantages and Limitations of CBPA Method

Advantages	**Limitations**
✓ Incorporates resource constraints into task planning, reducing the chance of assigning infeasible tasks.	✗ Increased computational complexity due to payload-aware evaluation and rollback logic.
✓ Selective rollback preserves executable tasks, avoiding unnecessary loss and improving scheduling robustness.	✗ Relies on consistent peer-to-peer communication, which may introduce delays in unstable networks.
✓ Achieves higher task utility and resource utilization compared to standard CBBA in dynamic environments.	✗ Added decision-making overhead may reduce responsiveness in real-time applications.

Although CBPA significantly enhances the feasibility of task allocation in resource-constrained scenarios by introducing a load-aware mechanism, it, like CBBA, is designed on the premise that all robots can function properly during task execution. The algorithm itself lacks a built-in fault-tolerant recovery mechanism to address hardware failures, software errors, or unexpected disconnections. Indeed, the original authors of CBPA explicitly stated in the conclusion of their paper that improving the fault-tolerance of the algorithm is a key direction for future work. A single robot's unexpected failure can disrupt the sequence of tasks assigned to it, and without an effective replanning strategy, such disruptions may jeopardize the successful completion of the entire team's mission.

3.2.2 Ant Colony Optimization and Bat Algorithm

Ant Colony Optimization (ACO) [37] is a distributed heuristic search method inspired by the behavior of ants in nature. Ants release pheromones in the process of looking for food. When the pheromone concentration in a particular path is very high, the possibility that subsequent ants will choose this path is greater. Ant colonies can gradually optimize the path through the accumulation and evaporation of pheromones in multiple iterations, thereby discovering a path scheme close to the global optimal solution.

Due to its inherent distributed characteristics, adaptive ability, and global optimization ability, ACO is widely used in multi-robot scheduling tasks. When each "artificial ant" moves between nodes, it selects the next moving node with a certain probability based on the local heuristic information (usually the reciprocal of the distance) and the global pheromone concentration. This not only realizes local exploration but also guides the overall convergence.

To enhance ACO, researchers propose a fusion strategy combining ACO with the Bat Algorithm (BA) [38]. The bat algorithm, as an optimization method inspired by bionics, simulates the behavior of bats using echolocation for prey search. It has good global search ability and dynamic environment adaptability. It is capable of resolving the task allocation conflicts generated by the independent planning of each robot in a distributed framework, and ultimately reaching a consensus to achieve a globally conflict-free optimal allocation scheme.

In the first stage of the fusion method, the ACO algorithm is used for the initial task allocation of the multi-robot system. Specifically, first, the pheromone values of all edges in the correlation graph between the robot and the task are initialized, and the heuristic information is calculated to reflect the attractiveness of the task. Each robot selects tasks based on the local pheromone concentration and heuristic information according to a certain probability, and the probability is defined as:

$$p_{ij} = \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum_{k \in allowed}(\tau_{ik})^{\alpha}(\eta_{ik})^{\beta}} \tag{3.2}$$

In the above formula, p_{ij} represents the probability that the robot selects the task j; τ_{ij} is the pheromone concentration on the path between the robot and the task, representing the cumulative intensity of the historical experience. The more pheromones there are, the more times this path has been chosen before, and it tends to be regarded as a high-quality path. η_{ij} is a heuristic function value, usually inversely proportional to the distance from the task to the robot. It can also be defined based on factors such as task priority and energy consumption, representing auxiliary information for immediate decision-making. The parameters α and β are adjustable weight coefficients used to control the degree of influence of pheromones and heuristic information in decision-making: when α is large, decisions rely more on historical experience (pheromone concentration); When β is larger, decisions tend to rely more on the current heuristic evaluation value (such as prioritizing tasks over shorter distances). The denominator part of the formula is normalized to ensure that after the summation of all the unfinished optional tasks k, the sum of the selection probabilities of each task is 1. The logical progression of this initial task allocation stage is summarized in Figure 3.7.

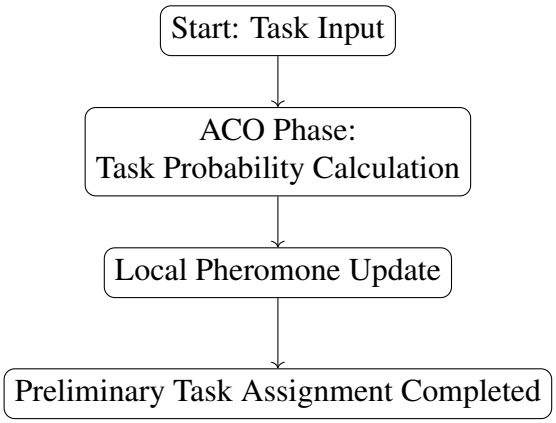

Fig. 3.7 First phase: preliminary task assignment process based on ACO.

In the second stage of the fusion method, the Bat Algorithm (BA) [39] is introduced to optimize and restore the initial task allocation generated by ACO. By simulating the behavior of bats using echolocation for prey search, BA has the dual capabilities of rapid jump search and local fine search, which can effectively compensate for the deficiency of ACO in dynamic environment adaptability. In the Bat Algorithm (BA), the search behavior of each individual is jointly driven by the dynamic updates of frequency, speed, and position. First of all, the frequency f_i is randomly generated and the formula is as follows:

$$f_i = f_{\min} + (f_{\max} - f_{\min}) \times \text{rand}(0,1) \tag{3.3}$$

f_i control of the individual search range, its scope is limited to $[f_{\min}, f_{\max}]$ range. Subsequently, the individual's speed is updated according to the following formula:

$$v_i^{(t)} = v_i^{(t-1)} + (x_i^{(t-1)} - x_*) f_i \tag{3.4}$$

If the current position $x_i^{(t-1)}$ deviates from the current iteration of the optimal location x_* and the velocity $v_i^{(t)}$ towards the adjustment of the direction of the optimal solution, to speed up the convergence process. Finally, the position update of the individual is given by the following formula:

$$x_i^{(t)} = x_i^{(t-1)} + v_i^{(t)} \tag{3.5}$$

That is, based on the current speed, move along the adjusted direction, thereby achieving continuous iterative optimization of the position [38]. To further enhance the local exploitation capability of the algorithm, a local search mechanism is triggered based on the pulse rate r_i. When a generated random number exceeds r_i, a new local solution is generated in the proximity of the current global best solution x_*:

$$x_{new} = x_* + \varepsilon A^{(t)} \tag{3.6}$$

where ε is a random number in the range $[-1,1]$ and $A^{(t)}$ is the average loudness of the bat population at the current timestep [23]. Since multi-robot task allocation is inherently a discrete optimization problem, these continuous variables must be discretized to form a valid assignment matrix. This process is governed by the following mapping and clamping functions:

$$S(v_i^{(t)}) = \frac{1}{1+e^{-v_i^{(t)}}}, \quad x_{ij} = \text{clamp}(x_i^{(t-1)} + v_i^{(t)}) \tag{3.7}$$

Furthermore, to simulate the convergence toward a target, the loudness A_i decreases while the pulse emission rate r_i increases as iterations progress, transitioning the swarm from exploration to exploitation. The update rules for these parameters are defined as:

$$A_i^{(t+1)} = \alpha A_i^{(t)}, \quad r_i^{(t+1)} = r_i^{(0)}[1-e^{-\gamma t}] \tag{3.8}$$

where α and γ are constants such that as $t \to \infty$, the system stabilizes around its optimal state.

These individual movement and parameter update rules provide the optimization engine for the **Consensus Phase**. In this stage, the BA is specifically introduced to resolve the task allocation conflicts that arise during the preceding ACO stage [39]. Each robot broadcasts its local task bundle constructed via ACO to its neighbors. The system then integrates these distributed bundles into a global task allocation matrix, which is treated as a potential "solution" (i.e., the position of a "bat") within the BA search space. By iteratively applying the BA search process, the system identifies an allocation matrix that maximizes global benefit while strictly satisfying all operational constraints. This finalized, conflict-free task plan is then issued to the entire robot fleet for execution. The complete dual-stage process, integrating ACO bundling and BA consensus, is depicted in Figure 3.8.

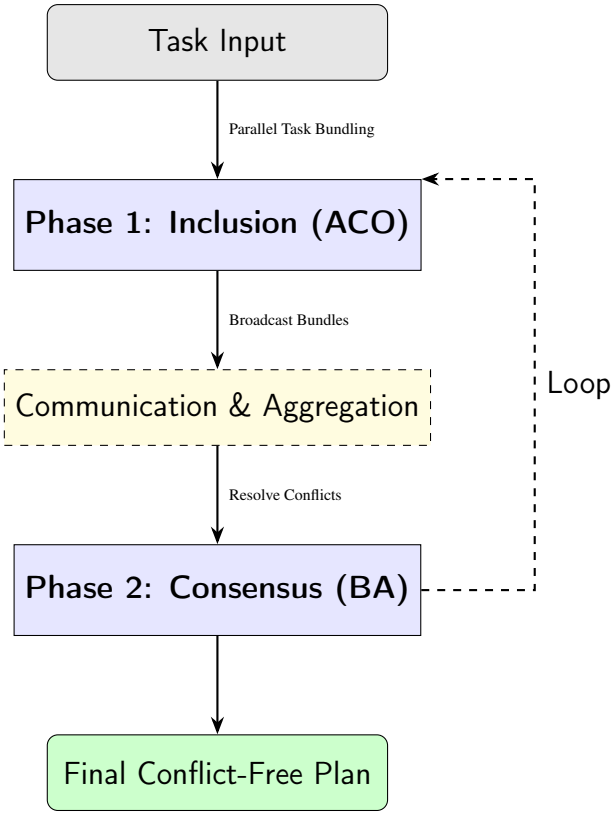

Fig. 3.8 Two-stage task allocation process based on ACO and BA

3.2.3 Reactive Re-auctioning Recovery Protocol (R3P)

R3P passively reallocates unfinished tasks of a failed robot via a localized re-auction, designed as a pluggable module atop CBPA/ACO+BA; see Figure 3.9 and Figure 3.10 for the flow.

FT-CBPA

In the framework design, the principles of inheritance and extension were strictly followed. The underlying task allocation logic of the framework completely retains the core mechanism of the classic consensus-based payload allocation algorithm (CBPA), including its distributed bundle construction, cost assessment based on payload constraints, and the basic process of resolving conflicts through consensus. Based on this solid foundation, we integrated a set of standard reactive re-auctioning recovery protocols into it.

Concretely, failure triggers isolation of orphan tasks and a one-shot local re-auction that re-inserts them into CBPA's plan (Figure 3.9).

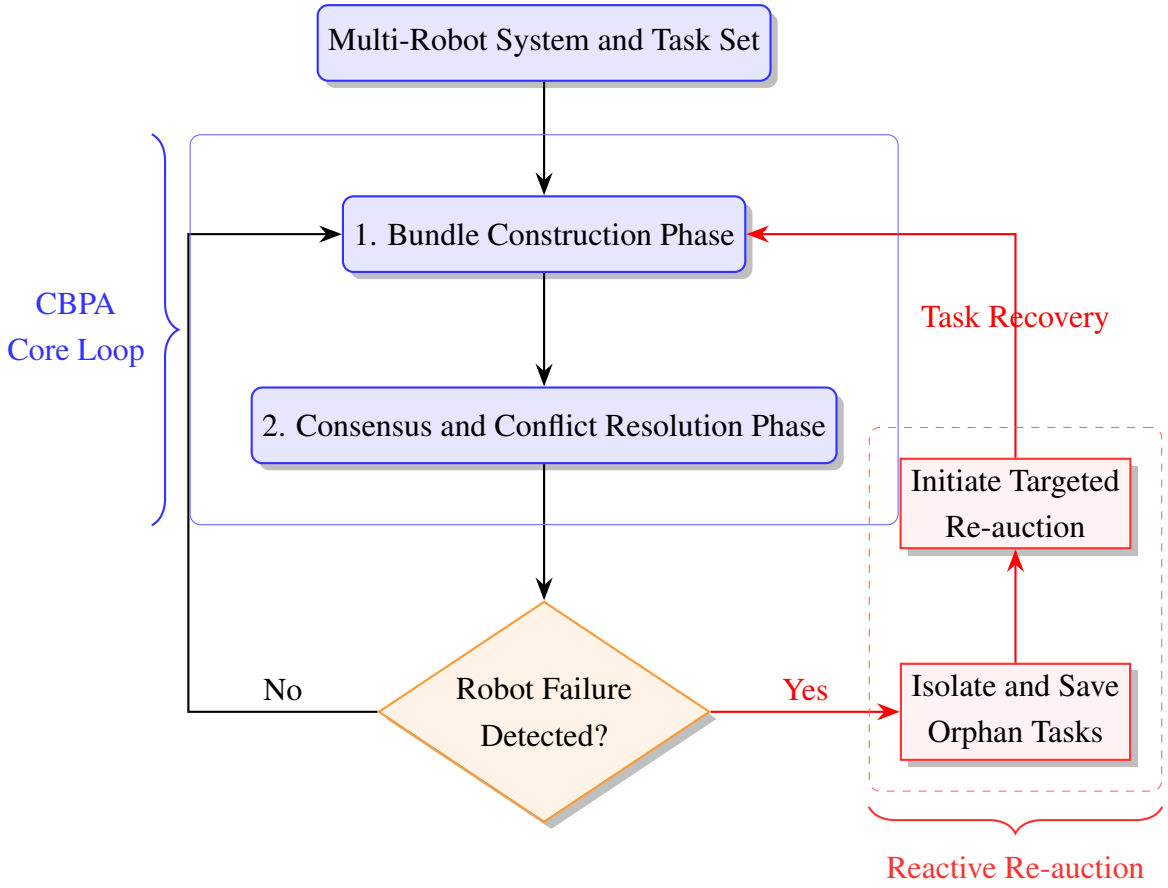

Fig. 3.9 CBPA-based Reactive Re-auction Recovery Framework Flowchart

FT-ACO+BA

The two-stage framework of ACO+BA proposed by Zitouni et al [40] is an advanced and distributed multi-robot task allocation method. It was found that its original architecture is fundamentally centered on planning, and it implicitly assumes a perfect execution environment. This framework lacks a native protocol to handle unexpected physical failures that occur in real time during task execution - such as hardware failures, communication losses, or path blockages. To fill this important research gap, this study integrates a lightweight reactive recovery protocol into the ACO+BA framework. The reuse of the bat algorithm (BA) module is carried out for this purpose. We transform the BA from a “task pre-consensus tool" that is only used to handle conflicts between initial task proposals to a “fault post-quick redistribution tool". The powerful optimization ability of BA can be dynamically invoked during task execution to solve a new emergent optimization problem: how to most efficiently redistribute the “orphan tasks" from the faulty robot to the remaining healthy robot team, taking into full consideration their real-time and dynamic states.

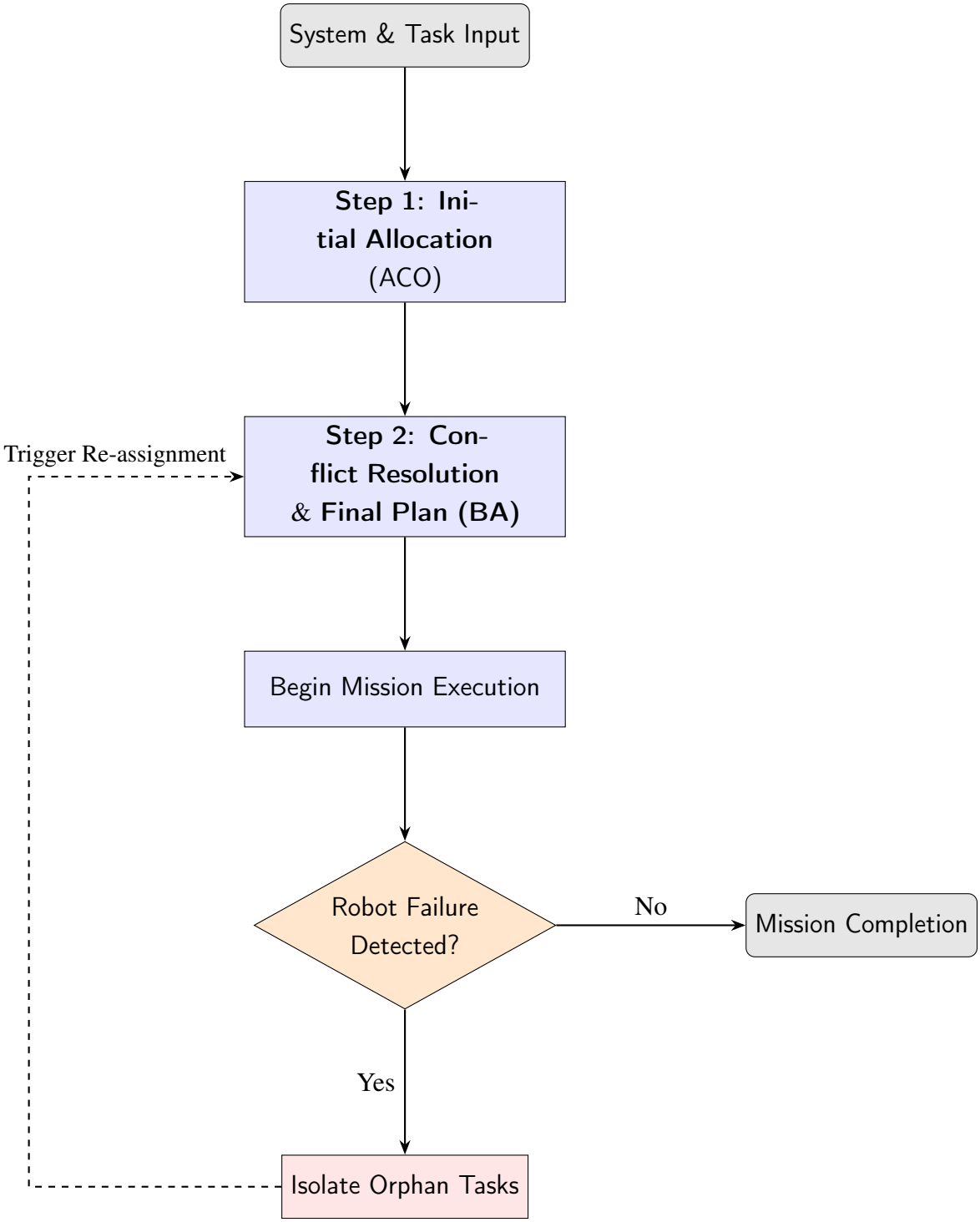

Fig. 3.10 Logical flow of the integrated FT-ACO+BA architecture with R3P protocol

In conclusion, the R3P protocol proposed in this study successfully provides CBPA and ACO+BA with the ability to handle physical failures through its unified design and deep integration with the scheduling core. Its core advantage lies in its high modularity and reusability, which clearly decouple the logic of fault recovery from the logic of task scheduling. This ensures the high-performance operation of the scheduling core without faults while providing critical system resilience. Of course, as a passive recovery strategy, this protocol

still has an inherent delay from the occurrence of a failure to the completion of re-allocation, and the results of local re-auction may not be the theoretically optimal global solution. Nevertheless, it provides a clear and efficient technical path for achieving a practical balance between performance and resilience.

3.3 Comparison of Fault-Tolerant Scheduling Methods

In the previous section, we thoroughly analyzed the four representative methods in native and integrated fault-tolerance architectures. In order to systematically evaluate their applicability in the modern intelligent warehousing environment and ultimately answer the core research question of this chapter - that is, which architecture is superior between native and integrated - this section will conduct an in-depth analytical discussion based on the comprehensive comparison framework presented in Table 3.7, from multiple key dimensions.

Table 3.7 Comparison of Fault-Tolerant Scheduling Architectures

Evaluation Criterion	**MRPF**	**BFTC**	**FT-CBPA**	**FT-ACO+BA**
High-Level Architecture	Centralized	Decentralized	Decentralized	Decentralized
Fault-Tolerance Philosophy	Native	Native	**Integrated**	**Integrated**
Primary Fault Type Handled	Simple Physical Failures	Byzantine Failures	Simple Physical Failures	Simple Physical Failures
Recovery Mechanism	Nearest available heuristic	Timeout & re-allocation via global consensus	Localized re-auction of "orphan tasks"	Localized re-optimization of "orphan tasks"
Recovery Speed / Latency	Very High (Low Latency)	Very Low (High Latency)	Medium	Medium
Communication Overhead	Low	Very High	Medium	Medium
Scalability	Low (Limited by controller)	Medium (Limited by consensus)	High	High
Suitability for Warehouses	Limited (Single point of failure)	Low (High latency, mismatch)	High (Balances performance)	High (Balances performance)

Table 3.7 clearly reveals the fundamental differences between the two design routes. MRPF and BFTC, as representatives of the native architectures, have inherent fault-tolerant mechanisms: The centralized architecture of MRPF enables recovery through a simple 'nearest replacement' heuristic rule, while the distributed architecture of BFTC relies on complex global consensus. This deep coupling in design also determines that the types of faults they mainly handle are highly specialized - the former is for simple physical faults, and the latter is for more complex Byzantine faults. In contrast, the two integrated methods we proposed (FT-CBPA and FT-ACO+BA) demonstrate design flexibility. By integrating a unified recovery protocol on a high-performance distributed scheduling core, they gain the ability to handle simple physical faults without changing their original advantages. Their recovery mechanism - 'local re-auction/re-optimization' - is a modular, post-trigger solution, in stark contrast to the built-in mechanisms of the native architectures.

Our integrated solution demonstrates a more balanced approach: under normal operation, its communication overhead remains consistent with the original algorithm, while a fault triggers only a localized and short-lived communication surge. This design achieves more

efficient resource utilization. The advantage becomes particularly clear in terms of scalability: MRPF is constrained by the physical bottleneck of its centralized controller, while BFTC suffers from increasing consensus delays as the network size grows. By contrast, our two integrated methods, built on a distributed core and localized recovery mechanism, exhibit superior scalability.

Under warehouse-style constraints, our comparison favors integrated routes (Table 4.31); Chapter 5 builds on this with SPA-CBPA. It allows us to first select a scheduling core that delivers optimal performance in fault-free conditions (e.g., CBPA) and then augment it with a plug-and-play modular protocol to address the most likely and impactful physical failures encountered in real-world scenarios.

Through the theoretical analysis and comparison in this chapter, we have highlighted the advantages of the integrated fault-tolerant architecture in terms of both design philosophy and performance trade-offs. Nevertheless, these theoretical deductions require validation through experimentation. In the following chapter, we will conduct a series of high-fidelity simulation experiments to quantitatively assess our conclusions and further verify the practical performance of integrated schemes such as FT-CBPA.

Chapter 4

Comparison of Existing Approaches in Simulated Environment

4.1 Simulation Execution Platform

The experiments were conducted in the cloud on Google Colab™, bypassing the need for a local hardware setup. The specific instance utilized an Intel® Xeon® dual-core vCPU with 13 GB RAM, running Ubuntu 20.04 (Linux 5.10). The entire system is a purely algorithmic discrete-event simulation[1], implemented in Python 3.10 without relying on any physical simulation engine. To guarantee reproducibility, the source code and dataset are publicly hosted on GitHub and use a fixed random seed of 42. To ensure the reproducibility of the proposed research, the complete software architecture, including the simulation framework and algorithm implementations, is systematically organized and provided in Appendix A.

4.2 Warehouse Grid and Obstacle Setting

The simulation environment uses a warehouse scenario with a 10×10 grid. This grid is configured with obstacles to mimic real-world congestion: fixed shelf obstacles (like those in columns 4 and 5) and one fixed, passage-blocking obstacle are combined with other randomly generated obstacles (shown in red). Gray areas in the map represent these shelves. In Figure 4.1, the T_i symbol represents tasks, and the R_i symbol represents the robots in the figure.

4.3 Number of Robots and Tasks

The initial positions of the robot are fixed at the four corners of the warehouse (for example, coordinates [0, 0], [9, 0], [0, 9], [9, 9]). In contrast, the positions of the task are randomly distributed in the free areas of the grid.

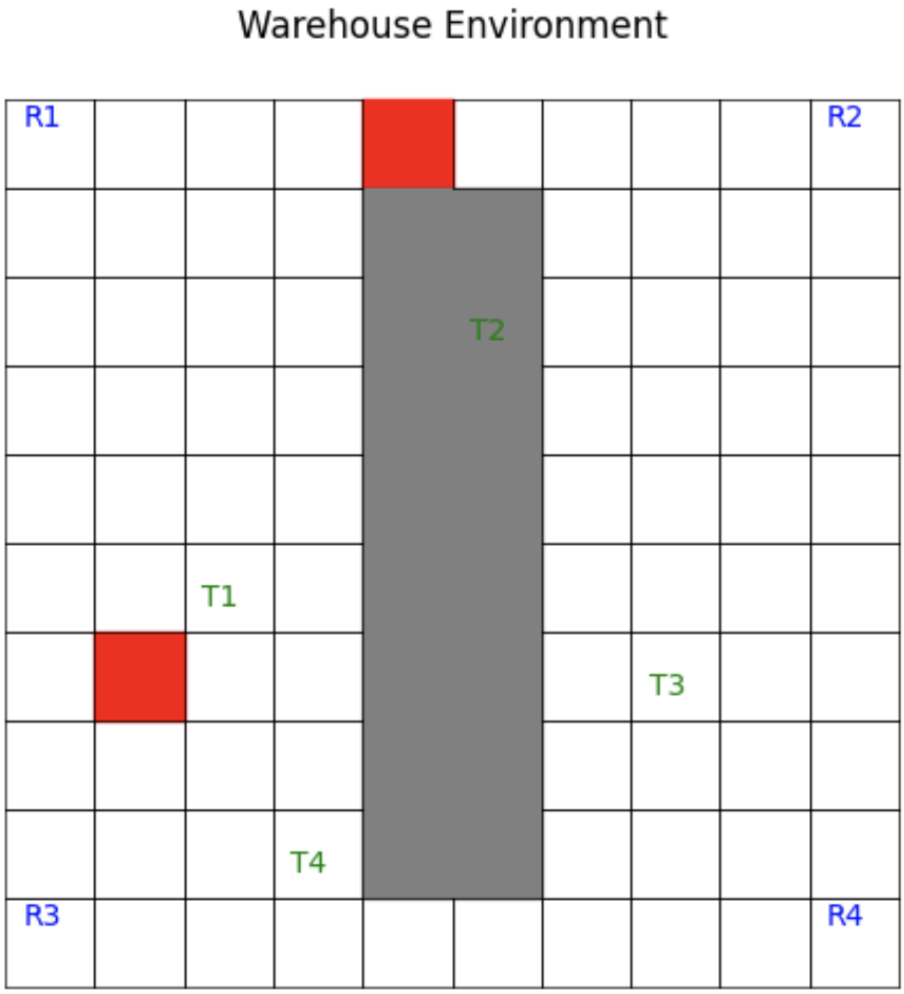

Fig. 4.1 Warehouse Environment

Rationale for Experimental Design and Fault Configuration. The structural selection of a 10×10 discrete grid layout combined with a deterministic single-robot failure injected at a designated operational step is intrinsically tied to the core research objectives. Because the primary focus of this dissertation is to explicitly investigate, profile, and evaluate post-fault recovery mechanisms and multi-agent resilience behaviors, the injection of a permanent node crash is a fundamental methodological requirement to observe the system under stress. Furthermore, as these initial simulation rounds serve as a preliminary comparative baseline and validation benchmark, restricting the evaluation to a compact spatial environment with a single localized failure node establishes a highly controlled testing envelope. This deliberate containment minimizes confounding environmental variables while remaining fully sufficient to clearly isolate, contrast, and mathematically expose the core performance characteristics, communication overhead spikes, and distinct task reallocation trade-offs inherent to the opposing scheduling architectures.

4.4 Implementation of two-dimensional (2D) Simulation Environment

To ensure a consistent basis for experimental comparison, we uniformly employed the A* search algorithm [34, 41] for all path-finding tasks, as none of the four core algorithms provided specific path-finding solutions.

4.4.1 Algorithm Implementation of Multi-Robot Preemptive Task Scheduling with Fault Recovery (MRPF)

MRPF Algorithm

We implement MRPF in a single Python module that follows the original pseudocode [19] line-by-line. This section documents the public entrypoints, data structures, and the mapping from Algorithm 4.1 to code. Reproducibility keys: fixed seed=42, grid=A* pathfinding, 3 execution nodes, uniform timing rules; hardware-agnostic 2D discrete-event simulation.

The MRPF scheduler is implemented in a single module with one main class `MRPFScheduler`. The public entrypoints and core helpers are summarized in Table 4.1.

Table 4.1 MRPF module structure and entrypoints

Component	Role / Signature
Scheduler class	`MRPFScheduler` — the main orchestrator.
Experiment driver	`MRPFScheduler.run_scheduling(max_time)` — outer loop calling one scheduling step per tick.
Core step	`MRPFScheduler.mrpf_scheduling_step()` — follows the paper's Algorithm 1 scheduling logic.
Core helpers	`execute_process()`, `preempt_process()`, `update_process_status()`, `get_priority_process()`, `allocate_robots_to_process()`, `handle_robot_failure()`, `resume_process()`, `plan_paths_for_process()`.

Table 4.2 summarizes entities and invariants. We use three execution nodes and a global process queue.

Table 4.2 Mapping of Conceptual Entities to Implemented Data Structures (MRPF)

Conceptual Entity	**Code**	**Description / Invariants**
Process	`self.process_queue`	Each item holds {`id`,`tasks`,`priority`,`status`,`robots_assigned`,`current_task_index`,...}. **Invariant:** status ∈ {`WAITING`, `EXECUTING`, `PREEMPTED`, `FINISHED`} (mutually exclusive).
Process Node	`self.process_nodes`	Fixed-size array representing execution slots. {`status`,`process`,`priority`}. **Invariant:** at most one process per node; if `status=FREE` then `process=None`.
Robot	`self.robots`	Individual robot {`id`,`current_position`,`velocity`,`status`,`current_process`,...}. **Invariant:** status ∈ {`FREE`, `BUSY`, `FAILED`}; if `BUSY` then `current_process` ≠ `None`.
Waiting Set	`get_priority_process('WAITING')`	Logical view over `self.process_queue`. **Invariant:** the function returns the unique highest-priority process in `WAITING` (ties broken deterministically).

Algorithm 4.1 is implemented by MRPFScheduler.mrpf_scheduling_step(); Table 4.3 gives the line-by-line mapping from the pseudocode and its corresponding Python implementation.

Algorithm 4.1: MRPF Scheduling function.

```
1  Function mrpf_scheduling(process_list)
2      process_choose = get_priority_process(process_list,"WAITING");;
3      if process_choose == NULL then
4          return;
5      end
6      smaller = 999;;
7      flag_execute = false;;
8      for i = 0; i < 3; i++ do
9          if process_node[i].status == "EXECUTING" AND;
10         get_process_status(process_node[i].process) == "FINISH" then
11             process_node[i].status = "FREE";
12         end
13         if process_node[i].status == "FREE" then
14             process_node[i].process = process_choose.process_id;
15             process_node[i].status = "EXECUTING";
16             update_process_status(process_choose.process,"EXECUTING");
17             execute(i, process_choose);
18             flag_execute = true;
19         end
20         else
21             if process_node[i].priority < smaller then
22                 index_smaller = i;
23                 smaller = process_node[i].priority;
24             end
25         end
26     end
27     if flag_execute == false AND process_choose.priority > smaller then
28         update_process_status(process_node[index_smaller].process,"WAITING");
29         preempt(process_node[index_smaller].process);
30         execute(index_smaller, process_choose);
31     end
```

Table 4.3 Line-by-line mapping between Algorithm 4.1 and the Python implementation

Lines	Python Code	Functionality
2–4	`process_choose = self.get_priority_process('WAITING') if not process_choose: return`	Picks the highest-priority waiting process via a helper function; returns if the queue is empty.
8–11	`self.update_process_status()`	Checks for completed processes and releases resources. This logic is encapsulated in `update_process_status()` and called at the start of the step for clarity.
12–18	`if self.process_nodes[i]['status'] == 'FREE': if self.execute_process(i, process_choose): flag_execute = True break`	On a free node, allocates and runs the process using the `execute_process()` helper.
19–24	`else: if self.process_nodes[i]['priority'] < smaller_priority: lowest_priority_node = i smaller_priority = self.process_nodes[i]['priority']`	Tracks the index and priority of the executing process with the lowest priority.
26–30	`if not flag_execute and process_choose.priority > smaller_priority: victim_process_id = self.process_nodes[lowest_priority_node] self.preempt_process(victim_process_id) self.execute_process(lowest_priority_node, process_choose)`	Pre-emption: preempts the victim via the `preempt_process()` helper and executes the new, higher-priority process.

When a higher-priority process appears, and no FREE node exists, `preempt_process()` stores `preemption_context={current_task_index, robots_assigned, preemption_time}` and releases robots, setting the process to `WAITING`; later `resume_process()` restores the context before re-execution. Upon failure, `handle_robot_failure(robot_id)` sets the robot to `FAILED`, removes it from `robots_assigned`, tries a FREE replacement, and re-plans paths via `plan_paths_for_process()`. If no replacement is available, the process

is set to `WAITING` and remaining robots are released (the node becomes FREE), guaranteeing progress when a robot is available again.

Let N be the number of process nodes (default 3), P the number of processes waiting/executing, and R the number of robots.

- One scheduling step (`mrpf_scheduling_step`): scans at most N nodes and selects max over waiting processes: $O(N+P)$.
- Preemption: constant-time victim selection within the step and context save/restore: $O(1)$ plus $O(|\text{robots\_assigned}|)$ for releasing robots.
- Failure handling: replacement search among FREE robots $O(R)$; re-plan path per assigned robot is delegated to the path planner.

Table 4.4 MRPF Parameters (implementation defaults)

Name	Default	Effect
`max_parallel_processes`	3	Upper bound of concurrent processes; larger values may increase congestion and preemption frequency.
Robot velocity	1.0	Affects `estimated_completion` and node release speed.
Failure rate (sim)	0.5%–1%/tick	Controls the frequency of injected failures in experiments.

We fix RNG seeds (`random.seed(42)`, `np.random.seed(42)`) before environment setup. We include three toy tests:

(T1) **Preemption correctness:** create two processes with priorities 4 and 2; start when all nodes are occupied by the low-priority process; assert that `preempt_process()` is invoked and the high-priority process enters EXECUTING within one step.

(T2) **Fault recovery:** force a BUSY robot to fail; assert that either a replacement is allocated and paths re-planned, or the process state flips to `WAITING` with robots released.

(T3) **No-deadlock invariant:** run for T ticks with periodic arrivals; assert that any EXECUTING process eventually transitions to `FINISHED` or `WAITING` (never stuck).

Experimental Results of MRPF

Scenario 1: number of tasks less than number of robots (2 tasks, 4 robots)

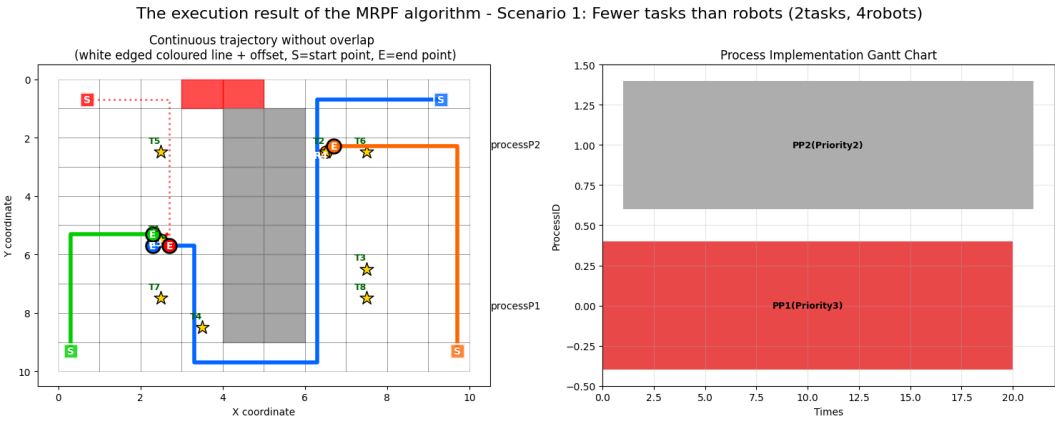

Fig. 4.2 Robot trajectories (left) and process Gantt chart (right) for Scenario 1

Table 4.5 MRPF — Scenario 1 (2 tasks, 4 robots): key performance indicators for a single run.

Metric	**Value**
Total tasks	2
Completed tasks	2
Completion rate (%)	100.0
Avg. completion time	20.50
Makespan	21.00

The experimental results under this configuration are illustrated in Figure 4.2. The trajectory graph (left) clearly presents the initial "multi-to-one" allocation strategy, where three robots (R_1, R_2, R_3) are assigned to high-priority process P_1, while P_2 is handled solely by R_4. A critical observation is the interruption of R_1's path near coordinates (3, 2), marked with a red cross, simulating a random hardware failure.

The Gantt chart (right) visually reveals the suspension and recovery mechanism of MRPF. A distinct temporal gap appears in the red block (P_1) around $T = 13$, corresponding to the phase where the process is forced to suspend due to insufficient resources after R_1's failure. Meanwhile, the grey block (P_2) maintains continuous execution, proving that low-priority tasks can run in parallel when resource contention is low. As P_2 finishes and releases R_4 at $T = 20$, P_1 immediately resumes execution in the Gantt chart, validating the effectiveness of the "nearest-idle-robot" dynamic replacement rule. Table 4.5 summarizes the KPIs for this scenario, where a 100% completion rate confirms the system's ability to maintain task continuity despite failures.

Scenario 2: number of tasks equals number of robots (4 tasks, 4 robots)

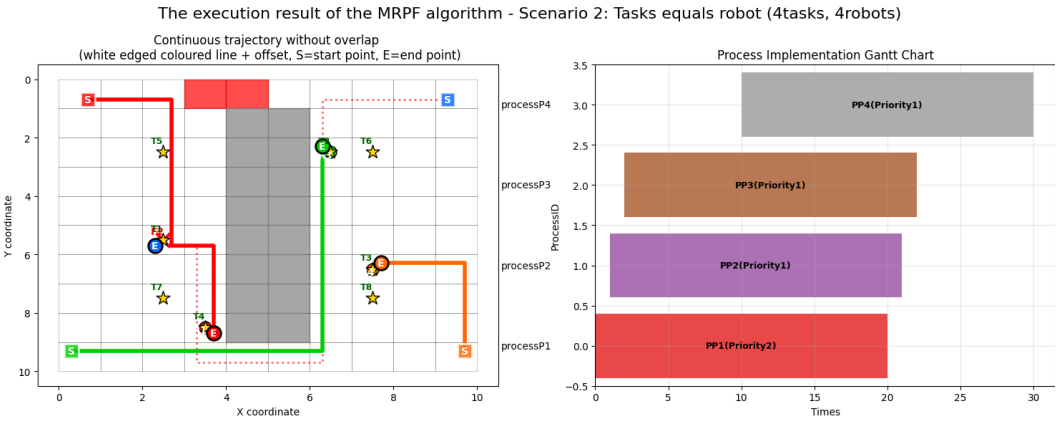

Fig. 4.3 Robot trajectories (left) and process Gantt chart (right) for Scenario 2

Table 4.6 MRPF — Scenario 2 (4 tasks, 4 robots): key performance indicators for a single run.

Metric	**Value**
Total tasks	4
Completed tasks	4
Completion rate (%)	100.0
Avg. completion time	23.25
Makespan	30.00

Under this balanced configuration, Figure 4.3 demonstrates increased scheduling flexibility. The trajectory graph shows more complex movement paths, yet spatial decoupling is maintained via the A^* algorithm.

The Gantt chart displays a hierarchical execution pattern. High-priority process P_1 is interrupted early due to R_2's failure (indicated by the first break in the colored blocks). Instead of triggering a global re-auction, the system waits for P_3, which has a shorter execution cycle, to finish. The Gantt chart shows that P_1 is reactivated immediately after P_3 releases its resources at $T = 10$, followed by P_4 filling the final gap. This "complete-release-reassign" step-like arrangement demonstrates how MRPF prioritizes the preemption rights of high-priority tasks. As shown in Table 4.6, the final Makespan is 30.00 units, aligning with the visual timeline in the Gantt chart.

Scenario 3: number of tasks more than number of robots (10 tasks, 4 robots)

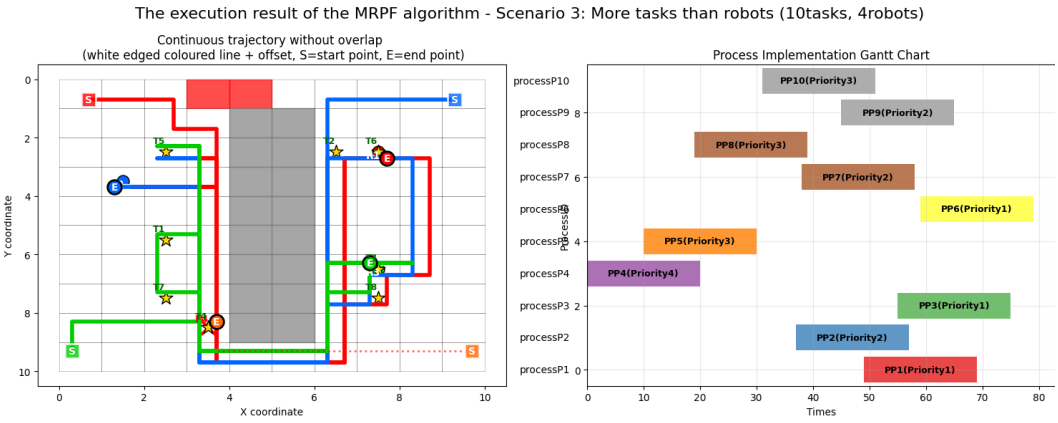

Fig. 4.4 Robot trajectories (left) and process Gantt chart (right) for Scenario 3

Table 4.7 MRPF — Scenario 3 (10 tasks, 4 robots): key performance indicators for a single run.

Metric	**Value**
Total tasks	10
Completed tasks	10
Completion rate (%)	100.0
Avg. completion time	54.30
Makespan	79.00

Under high-load pressure (10 tasks, 4 robots), Figure 4.4 illustrates the limit of MRPF's scheduling capacity. In the trajectory graph, dense paths reflect that robots must perform multiple round-trips to complete all tasks, yet the lack of overlapping paths confirms robust global collision avoidance.

The Gantt chart exhibits a significant "stepped" distribution. Due to severe resource scarcity, the 10 processes are forced into serial or small-scale parallel execution. The chart vividly records the long suspension interval of P_4 due to failure, followed by the sequential activation of P_5, P_8, and P_{10}. As indicated in Table 4.7, despite a 2.5:1 task-to-robot ratio and hardware failures, MRPF still achieves a 100% completion rate. Although the Makespan increases to 79.00, the compact block arrangement in the Gantt chart proves that resources are recycled to their maximum potential.

4.4.2 Algorithm implementation of Byzantine Fault-Tolerant Consensus (BFTC)

BFTC Algorithm

The core idea of the BFTC algorithm is to introduce a blockchain consensus layer, maintaining a distributed ledger in the multi-robot picking and delivery (MAPD) task. In this study, we followed the pseudo-code of the original paper (Algorithm 4.2) and replicated the core logic layer of BFTC in the Python simulation environment, including task allocation, A* collision avoidance, and consensus mechanism. To focus on the evaluation of algorithm performance and achieve rapid iteration, we simulated the behavior of Tendermint ABCI instead of relying on real blockchain hardware. Our implementation uses the Python 3 language and utilizes multiple scientific computing libraries, where NumPy is used for efficient matrix operations, and SciPy is used for performing optimization tasks. The entire project is designed as a reproducible simulation platform to verify the effectiveness of the BFTC algorithm in different scenarios.

Algorithm 4.2: BFTC (Byzantine Fault-Tolerant Consensus)

foreach *agent* a_i *in* $\mathscr{A}$ **do**
    a_i.path $\leftarrow$ loc(a_i);
    ; `// Start all agent processes, calling IndividualAgent(a_i, chain)`
end
while *true* **do**
    Add all new tasks, if any, to the task set $\mathscr{T}$;
    Put the new timestep on the chain;
    while $\exists a_i$ *not checked-in in the chain for the timestep* **do**
        ; `// system waits, with timeout`
        AdvanceTimestep(chain);
    end
    ; `// Move agents one step`
end

Algorithm 4.2: BFTC (continued)

```
Function IndividualAgent(a_i, chain)
    while true do
        success ← False;
        if a_i has no task then
            r ← QueryBestTaskAssignment(loc(a_i), chain);
            while r ≠ ∅ AND not success do
                usedElements ← QueryUsedElements(chain);
                path ← AStarPath(loc(a_i), r, usedElements);
                success ← DeliverPath(path, chain);
                if success then
                    assign a_i to r;
                    a_i.path ← path;
                    break;
                end
            end
        end
        if a_i has no task then
            if no r ∈ 𝒯 with g_r == loc(a_i) then
                while not success do
                    usedElements ← QueryUsedElements(chain);
                    safeEndpoint ← FindSafeEndpoint(loc(a_i));
                    path ← AStarPath(loc(a_i), safeEndpoint, usedElements);
                    success ← DeliverPath(path, chain);
                end
                a_i.path ← path;
            else
                while not success do
                    usedElements ← QueryUsedElements(chain);
                    safeEndpoint ← FindSafeEndpoint(loc(a_i));
                    path ← AStarPath(loc(a_i), safeEndpoint, usedElements);
                    success ← DeliverPath(path, chain);
                end
                a_i.path ← path;
            end
        end
        DeliverCheckIn(a_i, chain);
        while Not AdvancedTimestep(timestep, chain) do
            timestep ← GetTimestep(chain);
        end
    end
end
```

We implement BFTC as a Python simulation with a Tendermint-like consensus layer and an LO-MAPD loop. The system exposes one orchestrator (`BFTCSystem`) plus a minimal consensus simulator (`TendermintSim`) and a grid A* planner (`PathPlanner`). Table 4.8 lists public entrypoints with exact responsibilities. Our implementation follows Algorithm 1 in the paper and keeps the same outer/inner loop structure for agents, assignments, delivery, and check-ins.

Table 4.8 BFTC module structure and entrypoints.

Component / Function	**Role**
`BFTCSystem.run_simulation(max_timesteps)`	Main loop: per-timestep agent updates and time advancement.
`BFTCSystem.agent_step(agent)`	Individual agent logic: query/assign/plan/deliver and state transitions.
`BFTCSystem.query_best_task_assignment(agent)`	Global optimal pairing via Hungarian method [33] (per-timestep cache).
`PathPlanner.find_path(start,goal,dyn_obs)`	Grid A* search with dynamic obstacle injection.
`BFTCSystem.deliver_path(agent,path)`	Submit path to consensus: collision check $\rightarrow$ BFT vote $\rightarrow$ commit/rollback.
`TendermintSim.broadcast_and_commit(path,t)`	Voting and thresholding; on success, write time–location pairs.
`TendermintSim.check_collision(path,t)`	Spatio–temporal conflict detection against the replicated set.

Table 4.9 Mapping of conceptual entities to implemented data structures (BFTC).

Conceptual Entity	**Code**	**Description / Invariants**
Task	`BFTCSystem.tasks`	Each task: {`task_id,pickup,delivery,release_time,assigned,completed,assigned_agent`}. **Invariant:** once `completed=true`, it never returns to the pool.
Agent	`BFTCSystem.agents`	Robot state: {`agent_id,pos,path,full_trajectory,current_task,task_progress` $\in$ {`idle,to_pickup,to_delivery`}, `is_failed,is_byzantine,velocity`}. **Invariant:** if `is_failed` then `current_task=None` and `task_progress=idle`.
Ledger used elements	`TendermintSim.state.used_elements`	Replicated set of spacetime pairs (`timestep,cell`) reserved by committed paths. **Invariant:** append-only; committed pairs cannot be removed, preventing re-use collisions.
Consensus module	`TendermintSim.{check_collision,broadcast_and_commit}`	`check_collision` validates a path against `used_elements`; `broadcast_and_commit` counts votes and appends pairs on success. **Invariant:** commit requires yes-votes $\geq \lceil 2N/3 \rceil$.
Assignment cache	`BFTCSystem.query_best_task_assignment()`	Per-timestep Hungarian matching for idle agents and free tasks; cache cleared on `advance_timestep`. **Invariant:** each free task is assigned to at most one agent in the current step.
Path planner	`PathPlanner.find_path(start,goal,dyn_obs)`	Grid A* with dynamic obstacle injection; returns cell-by-cell plan. **Invariant:** returned path is geometrically feasible on the static grid before consensus checks.
System state	`BFTCSystem{ current_timestep,completed_tasks,path_rejects }`	Global counters and logs. **Invariant:** if a path is committed, all its pairs are appended to `used_elements` and the agent trajectory is updated synchronously.

Table 4.10 Line-by-line mapping between BFTC Algorithm (Algorithm 4.2) and the Python implementation

Lines	Python Code	Functionality

1–2	`def __init__(self, ..., start_pos, ...):`
	`self.pos = tuple(start_pos)`
	`self.full_trajectory = [tuple(start_pos)]`
		Initializes each agent's state, including its starting position and trajectory log.

4–10	`for t in range(max_timesteps):`
	`for agent in self.agents:`
	`self.agent_step(agent)`
	`self.advance_timestep()`
		Main simulation loop to process agents and advance the system's state.

13–14	`if agent.current_task is None:` `task = self.query_best_task_assignment(.)`	If an agent is idle, it queries for an optimal new task assignment.

15–22	`path = self.path_planner.find_path(...)`
	`if path and self.deliver_path(agent, path):`
	`agent.current_task = task`
	`agent.task_progress = "to_pickup"`
		Plans a path to the task's pickup location and submits it to the consensus layer for validation.

24–45	`elif agent.task_progress == "to_pickup":`
	`if agent.pos == task.pickup:`
	`# Plans path to delivery...`
	`elif agent.task_progress == "to_delivery":`
	`if agent.pos == task.delivery:`
	`task.completed = True`
	`agent.task_progress = "idle"`
		Handles task execution states: plans delivery path upon pickup, and completes task upon delivery.

46–51	`def advance_timestep(self):`
	`self.current_timestep += 1`
	`self._hungarian_cache = None`
		Advances the simulation clock and clears the task assignment cache for the next cycle.

We enforce safety by checking spatio–temporal conflicts before voting and committing. `TendermintSim.check_collision(path,t)` scans pairs $(t+\Delta, pos)$ against the replicated `used_elements`; on success, `broadcast_and_commit` performs majority voting (threshold $\geq \lceil 2/3 \cdot N \rceil$) and then appends pairs to `used_elements`. **Invariant:** once a path is committed, no later transaction can reuse the same pair, ensuring collision freedom by construction.

We simulate a network of N nodes with a Byzantine rate `byzantine_rate`; up to $\lfloor N/3 \rfloor$ nodes can behave arbitrarily. Malicious votes reduce `votes_yes`, but safety only depends on honest majority per-step validation; liveness is preserved when $\geq 2/3$ honest nodes exist.

Robot failures are modeled at the agent level (`Agent.is_failed`); upon failure, we release its task and re-enter the idle state for reassignment.

Let A be agents, T available tasks, N consensus nodes, and path length L.

- Assignment (Hungarian, square-padded): $O(\max(A,T)^3)$ per timestep (cached once per step).
- A* planning per path: $O(L \log L)$ on grid with consistent heuristic.
- Collision check: $O(L)$; commit appends L pairs to `used_elements`.
- BFT voting: $O(N)$ messages; threshold $\lceil 2N/3 \rceil$.

Table 4.11 BFTC parameters (implementation defaults).

Name	Default	Effect
`num_nodes` (`TendermintSim`)	4	Voting participants; larger N raises required quorum.
`byzantine_rate` (`TendermintSim`)	0.25	Fraction of malicious nodes for stress tests.
`threshold` (`TendermintSim`)	$2/3$	Commit rule; $\geq \lceil 2N/3 \rceil$ yes-votes required.
`MAP_SIZE`	10	Grid side length for the warehouse map.
`Agent.velocity`	2.0	Converts geometric length to real time for trajectories.
`max_timesteps`	100	Outer-loop guard for experiments.

We fix seeds via `random.seed(42)` and `np.random.seed(42)` before constructing the environment, ensuring deterministic task/agent initialization and obstacle placement. We provide three toy checks:

(T1) **Zero-collision invariant:** generate two agents, submit intersecting paths at same timestep; assert that `check_collision` rejects at least one before commit.

(T2) **BFT threshold:** set `num_nodes=4`, `byzantine_rate=0.25`; assert commit only when votes ≥ 3.

(T3) **Assignment optimality (per-step):** for a small $A{=}T{=}3$ instance, compare cost against Hungarian optimum; cached result must match the solver.

Experimental Results of BFTC

Scenario 1: fewer tasks than robots (2 tasks, 4 robots)

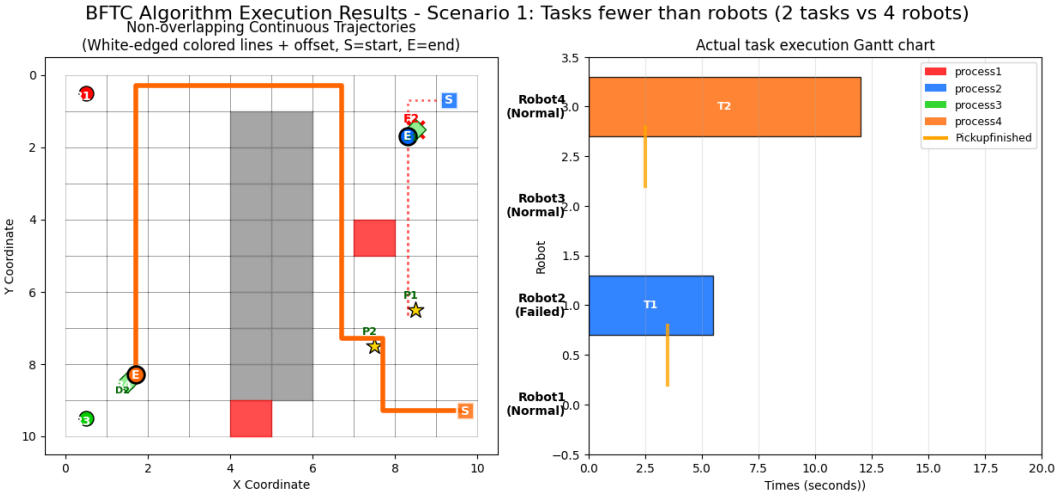

Fig. 4.5 Robot trajectories (left) and process Gantt chart (right) for Scenario 1

Table 4.12 BFTC — Scenario 1 (2 tasks, 4 robots): key performance indicators for a single run.

Metric	Value
Total tasks	2
Completed tasks	2
Completion rate (%)	100.0
Avg. completion time	14.17
Makespan	22.50

Figures 4.5 summarize the execution performance of BFTC under Scenario 1 (2 tasks, 4 robots). The left trajectory graph shows that, except for Robot 2, which failed in the 13th time slot, all the other robots completed their respective tasks along the collision-free path; the right Gantt chart further provides the timeline, marking the execution segments of T1 and T2 and the completion time of cargo pickup. Table 4.12 summarizes the core performance indicators: 100% task completion rate, average completion time of 14.17 seconds, and maximum completion time (makespan) of 22.50 seconds, which fully correspond to the 27 discrete walks in the log statistics.

Scenario 2: tasks equal robots (4 tasks, 4 robots)

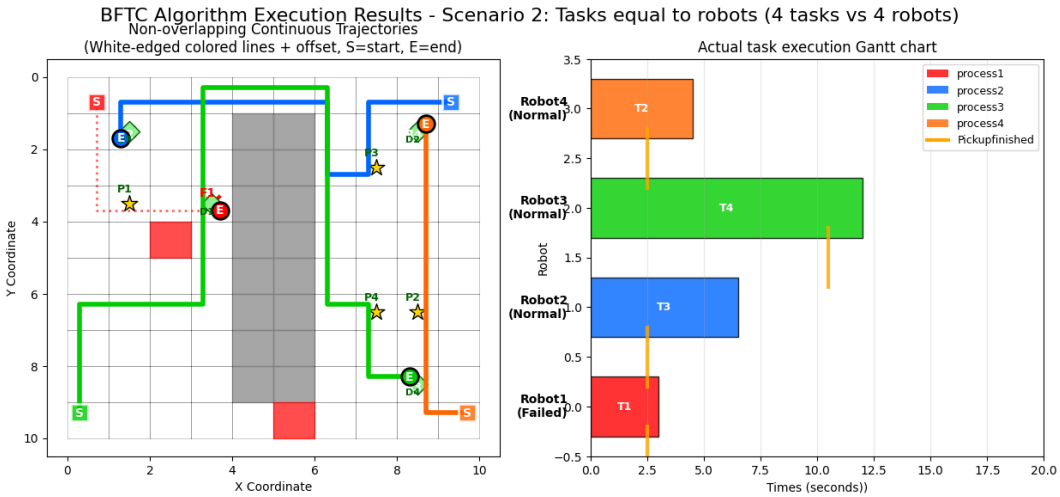

Fig. 4.6 Robot trajectories (left) and process Gantt chart (right) for Scenario 2

Table 4.13 BFTC — Scenario 2 (4 tasks, 4 robots): key performance indicators for a single run.

Metric	Value
Total tasks	4
Completed tasks	4
Completion rate (%)	100.0
Avg. completion time	27.83
Makespan	32.50

Figures 4.6 present the operation trajectory and Gantt chart of scenario 2 (4 tasks, 4 robots) under BFTC: The left figure shows that each robot is responsible for one pick-up and delivery path. After Robot 1 fails at step 7, it terminates (the red dotted line ends), and its task T1 is then taken over and completed by the remaining robots; in the right Gantt chart, the four colored blocks end successively, and the orange vertical line indicates the time when each task completes the pickup, with the termination line located at approximately 32.5 seconds. Table 4.13 summarizes the core performance indicators: All 4 tasks are completed, with an average completion time of 27.83 seconds and a maximum completion time (makespan) of 32.50 seconds.

By combining figures and the logs, it can be seen that BFTC still ensures the completion of all tasks even in the case of early robot failures, but multiple rounds of consensus and path re-proposal result in a slightly increased makespan compared to Scenario 1.

Scenario 3: more tasks than robots (10 tasks, 4 robots)

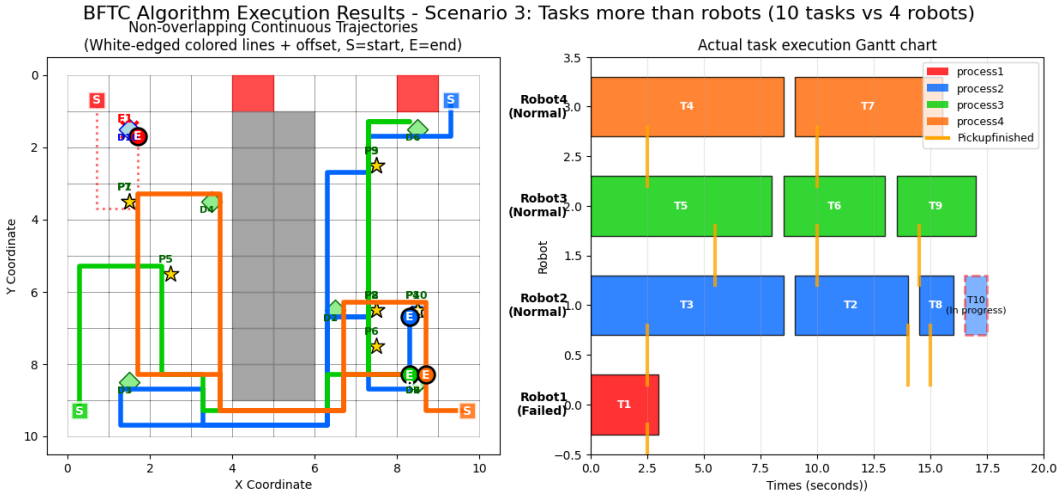

Fig. 4.7 Robot trajectories (left) and process Gantt chart (right) for Scenario 3

Table 4.14 BFTC — Scenario 3 (10 tasks, 4 robots): key performance indicators for a single run.

Metric	**Value**
Total tasks	10
Completed tasks	9
Unfinished tasks	1
Completion rate (%)	90.0
Avg. completion time	62.33
Makespan	62.50

Figures 4.7 illustrate the complete execution of Scenario 3 (10 tasks, 4 robots) under the BFTC framework. On the left trajectory graph, it can be seen that Robot 1 remained stationary after a failure at step 7, while the other three robots needed to take over continuously and complete the remaining tasks serially; the right Gantt chart shows that due to the large number of tasks compared to the number of robots, the strips exhibit obvious serial connections and overlaps, and there is still a short strip (T10) remaining at the end, corresponding to an unfinished task. The core indicators in Table 4.14 further quantify this phenomenon: of the 10 tasks, only 9 were completed (completion rate 90%), and the average completion time rose to 62.33 seconds, with the makespan reaching 62.50 seconds;

4.4.3 Algorithm implementation of ACO + BA

ACO + BA Algorithm

We intentionally omit the textbook ACO/BA background here. Our ACO+BA implementation reproduces the two–phase pipeline in Algorithm 4.3: (i) an ACO-based *inclusion*

phase to build local bundles, and (ii) a BA-driven *consensus phase* to select a conflict-free global assignment. The code exposes one driver and two core kernels. Table 4.15 lists the entrypoints.

Table 4.15 ACO+BA module structure and entrypoints.

Component / Function	**Role**
`run_aco(robots,tasks, ACO_ITERS,...)`	Inclusion phase: initialize τ, build bundles per robot, evaporate & reinforce pheromones.
`run_ba(bundles,POP,BA_ITERS,...)`	Consensus phase: initialize bat population (M_d, V_d, A_d, r_d), iterate frequency/velocity/loudness updates & local search.
`calc_gain(x)`	Fitness of a global assignment x (bat position); objective combines priority and A* path length.
`astar_path()`, `dist()`	Grid A* and shortest-path metric used by η and gain evaluation.

The simulation focuses on the logical verification of the ACO-based bundle construction and the BA-driven conflict resolution.

Algorithm 4.3: Steps of consensus phase run on each UAV.

foreach $u \in \mathscr{U}$ **do**
    send my survivor bundle to u;
end
wait until receiving all survivor bundles;
construct the matrix of survivor bundles as in Example 1;
initialize the bat population $M_d = \left[m_{pq}^d\right]$ and $V_d = \left[v_{pq}^d\right]$
($d = 1,\ldots,D,\ p = 1,\ldots,(|U| \times 3),\ q = 1,\ldots,(|S| \times 3)$);
initialize frequencies $F_d = \left[f_{pq}^d\right]$, pulse rates r_d and the loudness A_d;
while *not stopping criteria* **do**
    generate new solutions by adjusting frequency F_d using Equation 3.3;
    update velocities V_d and solutions M_d using Equations 3.4 and3.5, respectively;
    if *(*rand $> r_d$*)* **then**
        select a solution among the best solutions;
        generate a local solution around the selected best solution using Equations 3.6;
    end
    generate a new solution by flying randomly;
    discretize the search space using Equations 3.7, respectively;
    if *(*rand $< A_d$ **AND** $f(M_d) \leq f(M^*)$*)* **then**
        accept the new solutions;
        increase r_d and reduce A_d using Equations 3.8, respectively;
    end
    rank the solutions and find the current best solution M^*;
end
foreach $u \in \mathscr{U}$ **do**
    send my best solution M^* to u;
end
wait until receiving all the best solutions;
Find the global best solution and consider it as a solution to the MRTA problem;

To translate the abstract concepts into a functional simulation, several core data structures were defined:

- **Robot and Task:** Represented as Python `dictionaries`, storing attributes like ID, location (`loc`), and capabilities (`cap`).
- **Pheromone Matrix (τ):** Implemented as a 2D list of size *num_robots* $\times$ *num_tasks* for the ACO phase.
- **Bat Population:** In the BA phase, the population is a list of dictionaries. Each dictionary represents a "bat," containing its position `'x'` (a task-to-robot assignment vector), velocity `'v'`, loudness `'A'`, and pulse rate `'r'`.

Table 4.16 Core structures and invariants (ACO+BA).

Name / Type	Fields & Invariants
`robots:dict[int,{id,start}]`	Unique robot identifiers and starting cells. **Invariant:** `id` is unique; start cells are valid grid locations.
`tasks:dict[int,{id,location}]`	Pick-up (or service) locations. **Invariant:** once a task is selected in the final consensus solution, it is removed from the candidate set.
τ`(pheromone):2Dlist`	Dimension `num_robots` $\times$ `num_tasks`; updated by evaporation and reinforcement. **Invariant:** `clip_pos` enforces $\tau_{ij} \geq \varepsilon$ for numerical stability.
`bundles:list[list[int]]`	Per-robot candidate task sequences produced by ACO. **Invariant:** within a single ACO iteration, a task index is reinforced only for the robot that selected it.
`bat:{x,v,A,r}`	BA individual; `x` encodes a complete task–to–robot assignment vector; `v` is velocity; `A`/`r` are loudness/pulse rate. **Invariant:** `x` must represent a valid discrete assignment (no duplicate task).

The inclusion phase, executed by the `run_aco` function, is where each robot independently constructs a local task bundle. Table 4.17 provides a line-by-line mapping from the conceptual pseudo-code to the Python implementation. A key detail is the use of a clipping function, `clip_pos`, to ensure pheromone values remain above a small positive threshold, preventing numerical instability during updates.

Table 4.17 Line-by-line mapping for the Inclusion Phase of Algorithm 4.3 and the Python implementation

Lines	Python Code	Functionality

1, 3	`pher = [[PHER_INIT] * len(tasks) for _ in robots]`
	`bundles = [[] for _ in robots]`
	`max_dist = max(dist(r['loc'], t['loc']) ...)`
		Initialize the pheromone matrix (τ), empty task bundles for each robot, and other parameters before starting the main loop.

3	`for iteration in range(ACO_ITERS):`
	`  for ui, robot in enumerate(robots):`
	`  while avail:`
	`  # Calculate weights from tau & eta`
	`  tsel = random.choices(valid_tasks, ...)`
	`  bundles[ui].append(tsel)`
		Iteratively construct a candidate list of tasks (a "bundle") for each robot based on the probabilistic $\tau - \eta$ selection rule. This corresponds to the "construct a list of tasks" step.

3	`# Pheromone Evaporation`
	`pher[ui] = [clip_pos(p * (1 - RHO)) for p in pher[ui]]`
	`# Pheromone Reinforcement`
	`for task in bundles[ui]:`
	`  delta = (p_j(...)  - d_ij(...))`
	`  pher[ui][task['id']] += delta`
		Update the pheromone matrix by applying evaporation to all paths and then reinforcing the paths belonging to the newly constructed bundles.

3	`return bundles`	Return the constructed task bundles, which serve as the final output of the Inclusion Phase for the subsequent consensus algorithm.

Following the construction of local bundles, the Consensus Phase resolves conflicts to achieve a globally consistent allocation. This stage is implemented in the `run_ba` function. The quality of each potential global assignment (a “bat’s position") is evaluated by the `calc_gain` objective function.

Table 4.18 Line-by-line mapping for the *Consensus phase* of Algorithm 4.3 and our Python implementation

Lines	Python Code	Functionality

6	`pop = []`
	`for _ in range(POP):`
	`  bat = {‘x‘, ‘v‘, ‘A‘, ‘r‘}`
		Initialise bat population M_d, velocity V_d, loudness A_d, pulse rate r_d.

7	`best = max(pop, key=lambda` `b: calc_gain(b['x']))`	Rank initial population and set current best solution M^*.
8	`for it in range(BA_ITERS):`	Main loop *while not stopping criteria*.

9–10	`beta_d = random.random()`
	`freq = FMIN + (FMAX-FMIN) * beta_d`
	`bat['v'] = [v + (x_i - b_i)*freq]`
	`bat['x'] = clamp(x_i + v_i)`
		Adjust frequency F_d (Eq 3.3) and update V_d, M_d.

11–13	`if random.random() > bat['r']:` `  bat['x'] = clamp(b_i + N(0,1))`	Pulse-rate test; perform local search around the best solution.

15–17	`# random flight already`
	`# implicit in local search,`
	`clamp(...)`
		Random flight & discretisation.

18–19	`gain = calc_gain(bat['x'])`
	`S = 1/(1+e`$^{-\|v\|}$`)`
	`if gain > best_gain && rand<S:`
	`  best = deepcopy(bat)`
	`bat['A'] *= ALPHA_BA`
	`bat['r'] = bat['r']*(1-e`$^{-\gamma}$`)`
		Accept the new solution, then increase r_d and reduce A_d.

22	`best_gain & best updated inside loop`	Population re-ranking each iteration keeps the current best M^*.
26–27	`return best['x'], best_gain`	Output the global best assignment and its gain as the final MRTA solution.

Table 4.18 maps the core steps of the BA pseudo-code to the `run_ba` Python function. The implementation begins by initializing a population of “bats," where each bat’s position vector

represents a full task-to-robot assignment. The main loop then iteratively updates each bat's velocity and position using frequency modulation and performs a local search around the current best solution, as specified in the algorithm. The `calc_gain` function serves as the fitness evaluation to guide the search towards an optimal, conflict-free allocation.

Each candidate (either a bundle edge or a bat position) is evaluated with A* path length and feasibility checks. Infeasible moves receive prohibitive cost within `calc_gain`, so they are dominated and excluded from the final selection. The discretization routines `clamp` and the pheromone lower bound enforced by `clip_pos` prevent invalid assignments and numerical degeneration.

Let R denote the number of robots, T the number of tasks, L the average A* path length, I_{ACO} and I_{BA} the iteration counts, and P the BA population size.

- ACO inclusion: $O(I_{\mathrm{ACO}}RT)$ bundle construction plus path heuristics up to $O(L\log L)$ when invoked.
- BA consensus: $O(I_{\mathrm{BA}}P)$ fitness evaluations; each evaluation touches $O(T)$ assignment elements and, if needed, A* calls of cost up to $O(L\log L)$.

Table 4.19 ACO+BA parameters (implementation defaults).

Name (where)	Default	Effect
`ACO_ITERS` / `BA_ITERS`	50 / 25	Outer/inner iteration counts; trade off solution quality and runtime.
`PHER_INIT`, `RHO`, `EPS`	0.1, 0.1, 10^{-6}	Initial pheromone, evaporation rate, and lower bound (ε) enforced by `clip_pos`.
`ALPHA`, `BETA`	0.4, 0.6	Relative weights of pheromone vs. heuristic distance in selection.
`POP`	20	BA population size; too small risks premature convergence; too large increases runtime.
`FMIN`, `FMAX`	0.0, 2.0	Frequency modulation bounds governing exploration step sizes.

We fix seeds with `random.seed(42)` and `np.random.seed(42)` prior to any map/robot/task generation.

(T1) **Pheromone stability:** run `run_aco` for 10 iterations; assert $\min(\tau) \geq \varepsilon$ and $\max(\tau)$ remains bounded.

(T2) **Discrete validity:** in `run_ba`, assert that each bat position `x` encodes a valid one-to-one task–robot assignment (no duplicates).

(T3) **Gain monotonicity:** record `best_gain` across BA iterations; expect a non-decreasing sequence with possible plateaus.

Experimental Results of ACO + BA

Scenario 1: fewer tasks than robots (2 tasks, 4 robots)

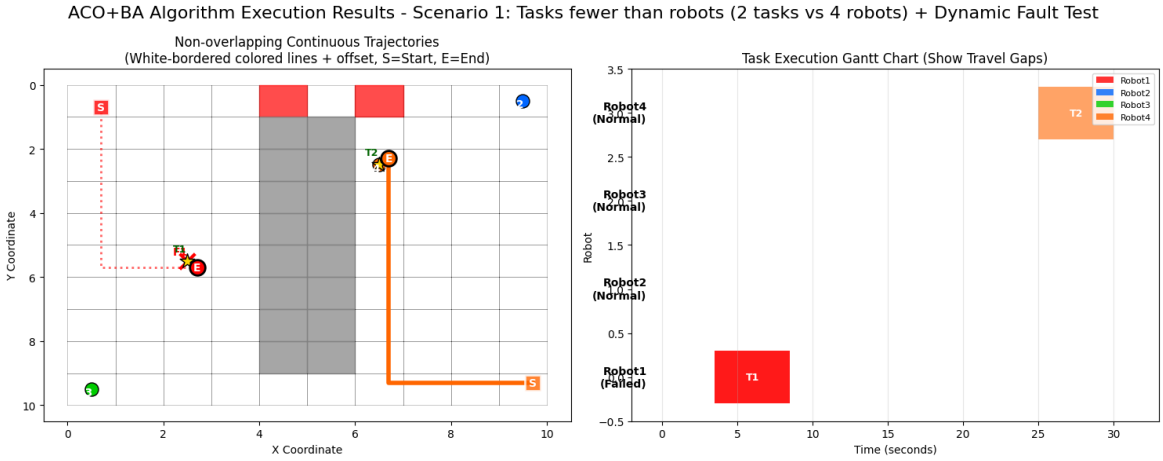

Fig. 4.8 Robot trajectories (left) and process Gantt chart (right) for Scenario 1

Table 4.20 ACO+BA — Scenario 1 (2 tasks, 4 robots): key performance indicators for a single run.

Metric	Value
Total tasks	2
Completed tasks	2
Completion rate (%)	100.0
Avg. completion time	19.25
Makespan	30.00

As shown in Figure 4.8 and Table 4.20, under the setting of "fewer tasks than robots" (2 tasks, 4 robots), ACO+BA still achieved a 100% task completion rate after introducing a run-time dynamic failure. From the trajectory graph, it can be seen that the planned trajectory of the system maintains a non-overlapping continuous form, with clear start and end points, and no path conflicts occurred within the failure time window; this indicates that the scheduling maintained a good spatial decoupling after the fault disturbance. The Gantt chart further shows the travel and waiting intervals, and the gap periods caused by rapid redistribution after the fault trigger can be clearly distinguished, but no task cascading delay was caused; the final makespan was 30.00 s, and the average completion time was 19.25 s. The results of this scenario indicate that in the resource-limited range where “the number of tasks is less than the number of robots", the re-allocation mechanism of ACO+BA can effectively limit the recovery delay and system-level side effects.

Scenario 2: tasks equal robot (4 tasks, 4 robots)

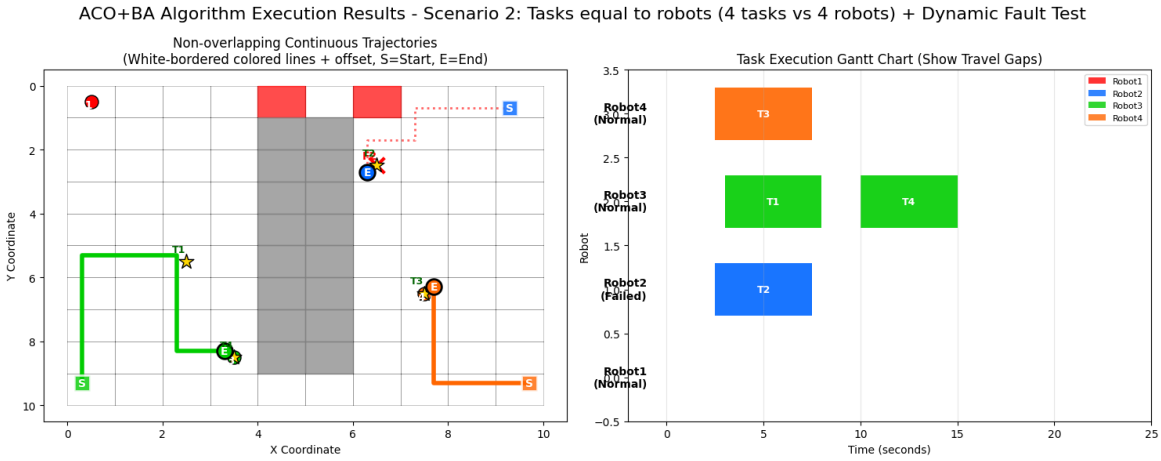

Fig. 4.9 Robot trajectories (left) and process Gantt chart (right) for Scenario 2

Table 4.21 ACO+BA — Scenario 2 (4 tasks, 4 robots): key performance indicators for a single run.

Metric	**Value**
Total tasks	4
Completed tasks	4
Completion rate (%)	100.0
Avg. completion time	9.50
Makespan	15.00

As shown in Figure4.9 and Table 4.21, under the balanced configuration where the number of tasks is equal to the number of robots (4 tasks, 4 robots), ACO+BA maintained a 100% completion rate after triggering a dynamic failure during the operation period. The system-level makespan was 15.00 s, and the average completion time was 9.50s. From the trajectory graph, it can be seen that the disturbances within the failure time window did not cause path congestion or interlocking. The trajectory presented a non-overlapping continuous form, indicating that the scheduling achieved effective decoupling in space. The Gantt chart clearly shows the travel and brief waiting intervals, and after reallocation, each robot basically achieved one-to-one task matching, thereby reducing queuing and resource idleness. Compared with Scenario 1 (where the number of tasks is less than the number of robots), in this scenario, while the resource utilization rate increased, the completion time significantly decreased (30.00s $\rightarrow$ 15.00s), indicating that under medium load, the fault recovery and task re-allocation mechanism of ACO+BA can compress the system-side delay without sacrificing stability.

Scenario 3: more tasks than robots (10 tasks, 4 robots)

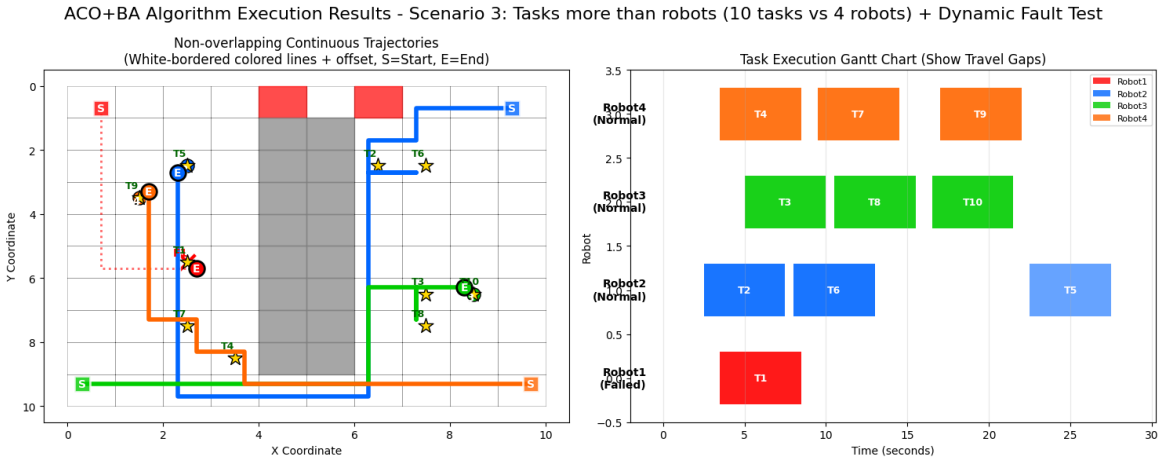

Fig. 4.10 Robot trajectories (left) and process Gantt chart (right) for Scenario 3

Table 4.22 ACO+BA — Scenario 3 (10 tasks, 4 robots): key performance indicators for a single run.

Metric	Value
Total tasks	10
Completed tasks	10
Completion rate (%)	100.0
Avg. completion time	14.85
Makespan	27.50

As shown in Figure 4.10 and Table 4.22, under the heavy load condition where the number of tasks is significantly greater than the number of robots (10 tasks, 4 robots), ACO+BA still achieved a 100% task completion rate after triggering a dynamic fault during the operation period. The system makespan was 27.50 seconds, and the average completion time was 14.85 seconds. The trajectory diagram shows that each robot took non-overlapping continuous paths, and no congestion or interlocking was observed during the fault time window. The Gantt chart shows obvious travel gaps and queuing segments, indicating that after fault recovery and reassignment, the system maintained a stable throughput through alternating serial and parallel operations and bounded waiting. Compared with Scenario 2 (4 tasks, 4 robots), the heavy load led to an increase in makespan (15.00 s $\rightarrow$ 27.50 s), but compared with Scenario 1 (2 tasks, 4 robots), with a makespan of 30.00 s, the current makespan is still under control. The average completion time (14.85 s) is between the two, reflecting the coexistence of parallel utilization before resource saturation and short-term queuing during the fault period.

4.4.4 Algorithm implementation of Consensus-Based Payload Allocation (CBPA)

CBPA Algorithm

Algorithmic background is covered in Chapter 3; this section documents code structure and reproducibility. Our implementation follows the two–phase CBPA procedure: a *payload bundle construction* phase that inserts tasks by marginal cost, and a *consensus* phase that trims coalitions and reconciles payload assignments. Table 4.23 lists the public entrypoints and their roles, aligned to the CBPA formalization.

Table 4.23 CBPA module structure and entrypoints.

Component / Function	Role (Algorithm anchor)
`calculate_insertion_cost_enhanced(robot,task)`	Computes marginal insertion cost C_{ij} and best position h using path-time difference Δt plus unmet-payload penalties α unmet_A $+ \beta$ unmet_B; used in bundle construction.
`allocate_payload_with_fault(coalition,task)`	Average payload allocation with failure awareness; mirrors Algorithm 1's loop until requirement is satisfied.
`simulate_route_...()`	A*-based travel time and arrival/finish estimation for bundles.
`consensus_update_with_fault(robots,tasks)`	Consensus trim: removes redundant robots from over-provided coalitions while preserving feasibility; reconciles winners and payload assignments.
`validate_and_mark_completed_tasks()`	Post-consensus state update (task status, coalition validity) with failure handling.

Table 4.24 Core structures and invariants (CBPA).

Name / Type	Fields & Invariants
`Robot`	`{id,(x,y),vel,A_init,B_init,A_remain,B_remain,bundle, is_failed}`; **Inv.** if `is_failed` then the robot neither receives new tasks nor contributes payload updates; completed tasks remain completed.
`Task`	`{id,(x,y),duration,A_req,B_req,status}`; **Inv.** `status=done` persists across consensus updates.
Conceptual matrices `X,T, L`	Winner/time/payload assignment views required by CBPA; implemented via per-robot bundles + coalition checks; **Inv.** feasibility test uses the sum of remaining payloads vs. requirements before commit.
`CollisionDetector`	Records time-stamped path segments for robots; **Inv.** no trimming step alters committed paths; conflicts are counted for metrics only.

The robot calculates the marginal cost [26] of inserting the task into its own path based on the current remaining load and the task requirements, and packages several tasks with the lowest costs into the local task sequence.

Algorithm 4.4: Payload Average Allocation

Input: Winners list of task j: x_{ij}, remaining payload of robots: r_i, requirements of task j: $\tilde{r}_j$

Output: Payload assignment matrix l_{ij}

$r'_i = \sum(r_i \cdot x_{ij})$, l_{ij};
if $r'_i \leq \tilde{r}_j$ **then**
    $l_{ij} = r_i$;
else
    while $\tilde{r}_j$ **do**
        $\bar{l} = \tilde{r}_j / \mathrm{len}\big(\{x_{ijk} \mid x_{ijk} = 1,\ k \in I\}\big)$;
        if $l_{ijk} < r_{ii} \leq \bar{l}$ **then**
            $l_{ijk} = r_{ii}$, $\tilde{r}_j = \tilde{r}_j - r_{ii}$, $x_{ijk} = 0$;
        else
            $l_{ijk} = l_{ijk} + \bar{l}$, $\tilde{r}_j = \tilde{r}_j - \bar{l}$;
        end
    end
end
return l_{ij};

Algorithm 4.4 addresses load balancing when multiple robots form an alliance to execute the same task j. It first computes the alliance's total remaining payload $r'_i = \sum(r_i \cdot x_{ij})$. If $r'_i \leq \tilde{r}_j$, each robot contributes all of its remaining payload r_i. Otherwise, an iterative procedure begins: in every loop, the unmet requirement $\tilde{r}_j$ is evenly divided among the current winners with $\bar{l} = \tilde{r}_j / |\{k \,|\, x_{ijk} = 1\}|$. When a robot's payload is exhausted, it is removed from the

alliance. The loop terminates once $\tilde{r}_j = 0$, ensuring the constraint $\sum_k l_{ijk} \geq \tilde{r}_j$ is always satisfied while maintaining a consistent global view of task demand.

Table 4.25 Line-by-line mapping between Payload Average Allocation (Algorithm 4.4) and the Python implementation

Lines	Python Code	Functionality

1–4	`def allocate_payload_with_fault(coalition, task):`
	`  active_coalition = [r for r in coalition if not r.is_failed]`
	`  total_A_available = sum(r.A_remain for r in active_coalition)`
	`  if total_A_available < A_req:`
	`  for r in active_coalition:`
	`  alloc[r.id]['A'] = r.A_remain`
		Initializes by identifying active robots and their total available payload. If insufficient, each robot allocates its entire remaining payload, mapping to the logic in lines 2-3.

5–12	`  else:`
	`  remaining_req = A_req`
	`  members = active_coalition[:]`
	`  while remaining_req > 0 and members:`
	`  equal_share = remaining_req / len(members)`
	`  for r in members:`
	`  if r.A_remain >= equal_share:`
	`  alloc[r.id]['A'] += equal_share`
	`  else:`
	`  alloc[r.id]['A'] += r.A_remain`
		Handles cases where the payload is sufficient. It iteratively calculates an equal share of the remaining requirement for active robots, distributing it until the task's needs are met, mirroring the 'while' loop logic.

14	`  return alloc`	Returns the final payload assignment dictionary, which serves as the assignment matrix L_{ij}.

Algorithm 4.5: Payload Bundle Construction of Robot i

Input: Robot time matrix $T_i(t-1)$, payload assignment matrices $L_i^a(t-1)$, $L_i^b(t-1)$

Output: $T_i(t)$, $L_i^a(t)$, $L_i^b(t)$

Update X_i, r_i, M_i, τ_i, b_i, p_i;
while $r_i^a > 0$ ***or*** $r_i^b > 0$ **do**
  Compute $C_{ij}, \forall j \in J$;
  $g = \min_{j \le N_t} C_{ij}$;
  if $0 < g < c$ **then**
    $q = \arg\min_{j \le N_t} C_{ij}$;
    $h = \arg\min_{n \le |p_i|+1} \{ S_i(p_i \oplus_n \{q\}) - S_i(p_i) \}$;
    if $m_{iq}^a > 0$ ***or*** $m_{iq}^b > 0$ **then**
      $x_{iqi} = 1, \quad t_{iqi} = t_{i,p_i,h}$;
      Compute and update L_i according to Algorithm 1;
    end
    if $m_{iq}^a = 0$ ***or*** $m_{iq}^b = 0$ **then**
      $i' = \arg\max_{x_{iqk}=1} t_{iqk} \cdot x_{iqk}$;
      $x_{iqi} = 1, \quad x_{iqi'} = 0$ Compute L_i;
      if $m_{iq}^a = 0$ ***or*** $m_{iq}^b = 0$ **then**
        $t_{iqi} = t_{i,p_i,h}$, and update L_i;
      end
    end
    Update M_i, τ_i, b_i, p_i;
  end
end

Algorithm 4.5 determines, at each bidding round, which new task robot i should insert into its path and maintains local state accordingly. It begins by refreshing the winner matrix X_i, remaining payload r_i and the current task sequence p_i. While $(r_i^a > 0) \vee (r_i^b > 0)$, it calculates the marginal insertion cost C_{ij} for every candidate task j, selects the minimum $g = \min_j C_{ij}$, and if $0 < g < c$ chooses the target task q and best insertion position h. Depending on whether q already meets its payload requirement, Algorithm 4.4 is invoked to update the payload matrix L_i; if not, the slowest robot currently assigned to q is replaced to shorten the task start time. The loop ends when both payload types are depleted, or no further tasks can be bid for, outputting the updated time matrix $T_i(t)$ and payload matrix $L_i(t)$.

Within the CBPA framework, Algorithm 4.5 repeatedly calls Algorithm 4.4 during bundle construction. Whenever a robot joins or leaves an alliance, Algorithm 4.4 re-averages the payload to keep the "payload satisfaction" and "minimum average task start time" objectives aligned. Through this complementary design, CBPA produces collision-free, requirement-compliant allocations even under continuous payload depletion.

Table 4.26 Line-by-line mapping between Payload Bundle Construction (Algorithm 4.5) and the Python implementation

Lines	Python Code	Functionality

2–3	`while iteration < MAX_ITER:`
	`  for robot in robots:`
	`  if robot.is_failed:  continue`
	`  for task_id, task in tasks.items():`
	`  if task.id in assigned_tasks:`
	`continue`
	`  if task.A_req > 0 and`
	`robot.A_remain <= 0:`
	`  continue`
		Corresponds to the main 'while' loop that continues as long as the robot has remaining payload. The nested loops iterate through active robots and unassigned tasks to find potential candidates for the bundle.

4–7	`insert_idx, inc_cost =`
	`  calculate_insertion_cost_enhanced(...)`
	`if inc_cost < best_cost:`
	`  best_cost = inc_cost`
	`  best_task = task.id`
	`  best_insert_idx = insert_idx`
		This block calculates the insertion cost (C_{ij}) for each task and finds the task with the minimum cost (g and q). The `calculate_insertio` function also determines the best position (h) to insert the task.

8–11	`if best_task is not None and best_task not`
	`in assigned_tasks:`
	`  robot.bundle.insert(best_insert_idx,`
	`best_task)`
	`  assigned_tasks.add(best_task)`
		After the best task to add is identified, this code block officially adds it to the robot's bundle (p_i) at the optimal insertion point. It sets the winner flag ($x_{iqi} = 1$) by adding the task to the `assigned_tasks` set.

Table 4.26 – continued from previous page

Lines	Python Code	Functionality

12–18	`task_coalitions, ... = consensus_update_with_fault(...)`
	`for r in removed:`
	`  if t_id in rob.bundle:`
	`  rob.bundle.remove(t_id)`
		While not a direct match, the `consensus_update_with_fault` function serves a similar purpose. It reviews coalitions and can remove a robot from a task (effectively setting $x_{iqi} = 0$) if others can handle the payload.

19	`for r in robots:`
	`  if not r.is_failed:`
	`  used_A = sum(tasks[tid].A_req`
	`  for tid in r.bundle)`
	`  r.A_remain = r.A_init - used_A`
		At the beginning of each main iteration, the robot's remaining payload (r_i) and other state variables (M_i, τ_i, b_i, p_i) are updated based on the tasks currently in its bundle (p_i).

Following each bundle-construction sweep, `consensus_update_with_fault` aggregates per-task coalitions, checks feasibility $\sum l_{ijk} \geq \tilde{r}_j$, and *removes redundant robots* while keeping the task feasible. This "trimming" is the practical mechanism that enforces the update rules in CBPA's Table I: payload-satisfaction dominates, then earlier start time τ_j breaks ties. The routine is fault-aware: failed robots do not contribute payload and are pruned from coalitions and bundles.

Travel times are computed from grid A* paths; each insertion recomputes arrival and finish times as in Eq. 3.1. A time-stamped path store enables pairwise spatio–temporal conflict checks used for reporting (collisions are counted as a metric; they do not alter coalition decisions).

Let R robots, T tasks, L average A* path length, and I the number of bundle-construction sweeps.

- Marginal insertion search per robot: $O(T)$ calls to `simulate_route` plus A* with cost up to $O(L \log L)$; worst-case $O(T L \log L)$.
- Coalition trim per task: $O(R)$ feasibility checks and removals.
- End-to-end per sweep: $O(RT L \log L + TR)$; over I sweeps gives $O(IRT L \log L)$.

These bounds reflect the constructive CBPA loop and are consistent with the two-phase design.

Table 4.27 CBPA parameters (implementation defaults).

Name	Default	Effect
`ALPHA_WEIGHT`, `BETA_WEIGHT`	1000.0, 1000.0	Weights for unmet payload penalties in C_{ij}; enforce strong preference for requirement satisfaction before travel-time improvements.
`VELOCITY_RANGE`	(2.0, 2.0)	Constant robot speed used for A*-derived travel time.
`A_PAYLOAD_RANGE`, `B_PAYLOAD_RANGE`	(5,15), (8,15)	Initial capacities; drive r_i^a, r_i^b and feasibility.
`A_REQ_RANGE`, `B_REQ_RANGE`	(0,10), (0,20)	Task requirements; determine demand vectors $\tilde{r}_j$.
`DURATION_RANGE`	(5.0, 10.0)	Service time δ_j added to arrival time for start/finish estimates.

We fix seeds via `random.seed(42)` and `np.random.seed(42)` before environment generation. Minimal tests:

(T1) **Feasibility monotonicity:** consensus trim never breaks feasibility ($\sum l_{ijk} \geq \tilde{r}_j$ remains true once achieved).

(T2) **DMG trend:** recorded C_{ij} values are non-increasing across consensus/bundle iterations until convergence (cf. Eq. 3.4).

(T3) **Fault robustness:** after forcing a robot failure post first task, completed tasks remain `done` and uncompleted tasks are reassignable.

Experimental Results of CBPA

Scenario 1: fewer tasks than robots (2 tasks, 4 robots)

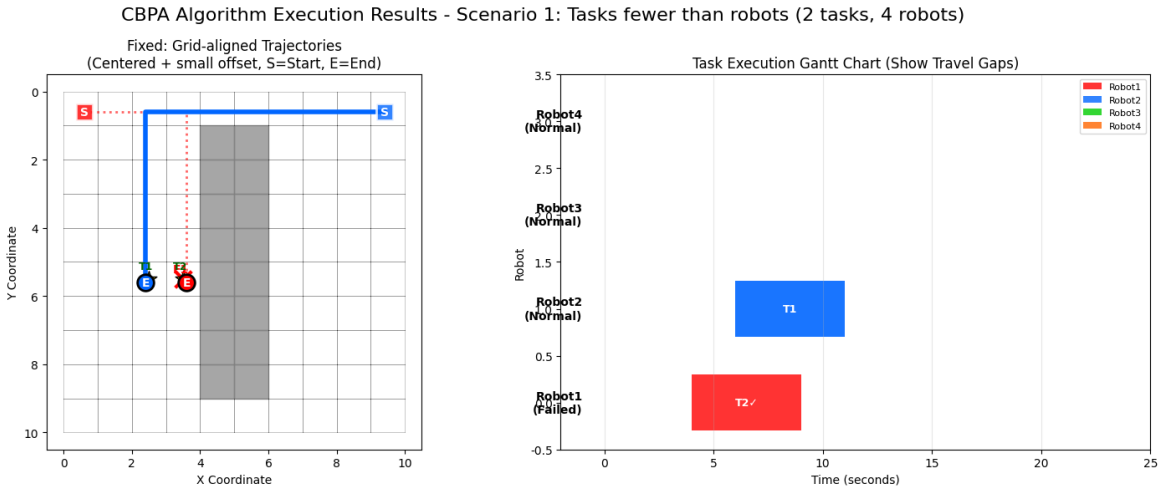

Fig. 4.11 Robot trajectories (left) and process Gantt chart (right) for Scenario 1

Table 4.28 CBPA — Scenario 1 (2 tasks, 4 robots): key performance indicators for a single run.

Metric	Value
Total tasks	2
Completed tasks	2
Completion rate (%)	100.0
Avg. completion time	11.00
Makespan	11.00

As illustrated in Figure 4.11 and Table 4.28, CBPA achieved a 100% task completion rate in the "fewer tasks than robots" scenario. The trajectory graph (left) shows that the four robots maintain clear spatial decoupling, with paths primarily distributed around the pickup and delivery points without any spatio-temporal conflicts. The Gantt chart (right) reveals that tasks T1 and T2 were executed almost entirely in parallel, resulting in an average completion time and a makespan of 11.00 seconds. These results indicate that under low-load conditions, the marginal cost-based insertion mechanism of CBPA can identify the most efficient robot-task pairings while ensuring optimal resource utilization.

Scenario 2: tasks equal robot (4 tasks, 4 robots)

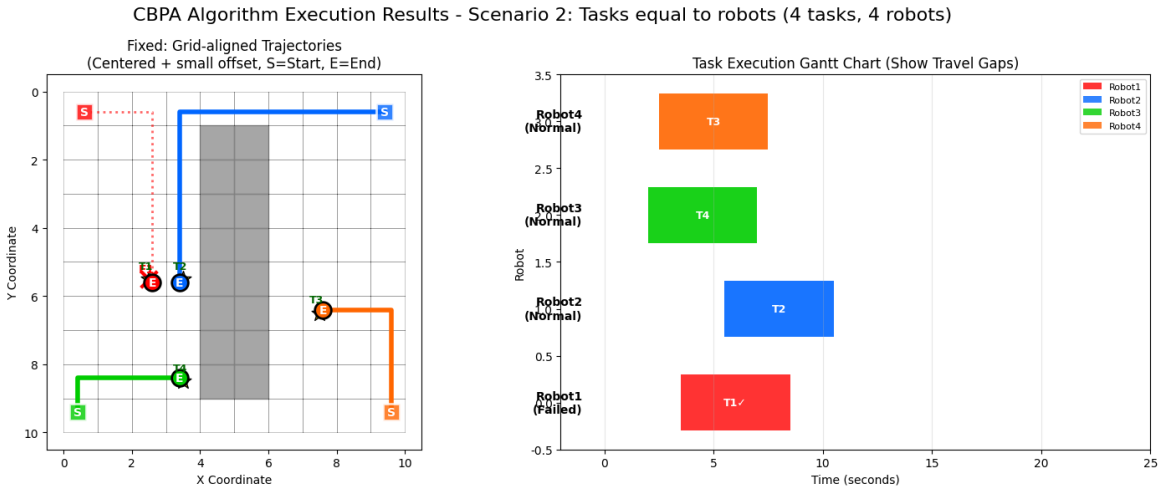

Fig. 4.12 Robot trajectories (left) and process Gantt chart (right) for Scenario 2

Table 4.29 CBPA — Scenario 2 (4 tasks, 4 robots): key performance indicators for a single run.

Metric	Value
Total tasks	4
Completed tasks	4
Completion rate (%)	100.0
Avg. completion time	8.33
Makespan	10.50

Figure 4.12 and Table 4.29 present the performance of CBPA under a balanced configuration. The system maintained a 100% completion rate, with the average completion time further optimized to 8.33 seconds. From the trajectory diagram, it is observed that each robot was assigned to a specific task bundle, following a collision-free path even when a dynamic failure was triggered. The Gantt chart shows that the consensus phase successfully resolved potential allocation conflicts, ensuring that the remaining robots could take over tasks from the failed node with minimal delay. The makespan of 10.50 seconds suggests that CBPA's "trimming" mechanism effectively balances the workload across the fleet.

Scenario 3: more tasks than robots (10 tasks, 4 robots)

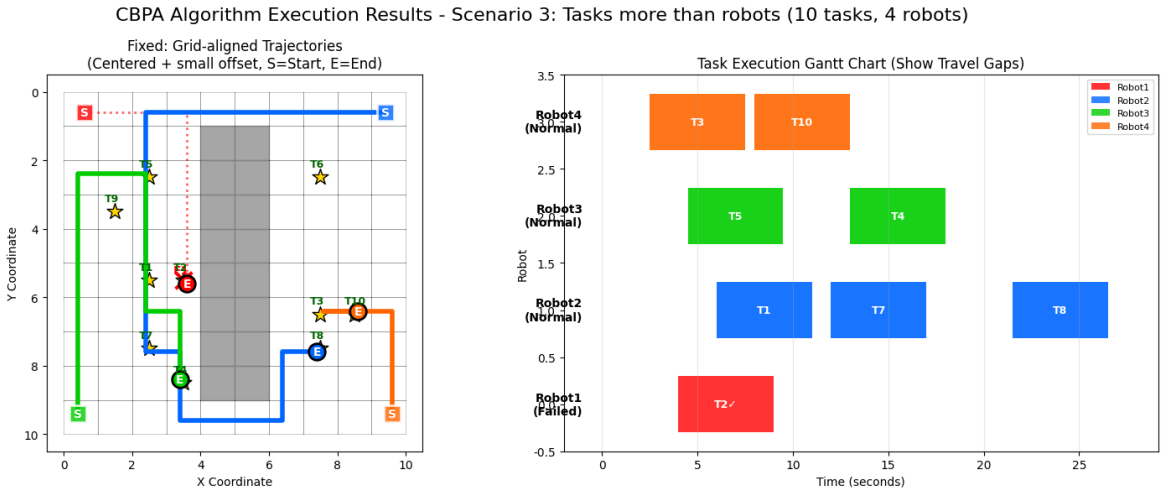

Fig. 4.13 Robot trajectories (left) and process Gantt chart (right) for Scenario 3

Table 4.30 CBPA — Scenario 3 (10 tasks, 4 robots): key performance indicators for a single run.

Metric	**Value**
Total tasks	10
Completed tasks	8
Unfinished tasks	2
Completion rate (%)	80.0
Avg. completion time	14.64
Makespan	26.50

Under the heavy-load condition (Scenario 3), the performance of CBPA is summarized in Figure 4.13 and Table 4.30. Unlike the previous scenarios, the completion rate dropped to 80%, with 2 tasks remaining unfinished within the simulation horizon. The trajectory graph shows highly complex and overlapping paths as the four robots attempted to handle 10 tasks serially. The Gantt chart illustrates significant queuing effects; following the failure of Robot 1 at step 7, the system load increased substantially, leading to a makespan of 26.50 seconds. This phenomenon demonstrates the scalability limit of the current CBPA implementation when the task-to-robot ratio is high and the available payload capacity is restricted.

4.5 Experimental Results Comparison of the Four Algorithms

Table 4.31 presents the overall performance of the four multi-robot task allocation algorithms (MRPF, CBPA, BFTC, and ACO+BA) evaluated in this study across three different load scenarios. These scenarios correspond to cases where the number of tasks is fewer than

(S1), equal to (S2), and greater than (S3) the number of robots. To ensure the robustness and statistical reliability, the key performance indicators reported in the table, such as average completion time (AvgT) and maximum completion time (Makespan), are calculated through 30 independent repeated experiments and reported in the form of “mean ± standard deviation".

Table 4.31 Four Algorithms — Core Performance Metrics Across Three Load Scenarios

Algorithm	**Scenario**	**Total**	**Done**	**Rate (%)**	**AvgT**	**Makespan**	**n**
ACO+BA	S1	2	2	100	11.38 ± 4.17	13.80 ± 6.91	*30*
	S2	4	4	100	9.50 ± 0.00	15.00 ± 0.00	*30*
	S3	10	10	100	16.36 ± 4.79	34.77 ± 15.74	*30*
BFTC	S1	2	1	50	17.00 ± 0.00	17.00 ± 0.00	*30*
	S2	4	3	75	14.83 ± 0.00	21.50 ± 0.00	*30*
	S3	10	4	40	18.50 ± 0.00	30.50 ± 0.00	*30*
CBPA	S1	2	2	100	12.00 ± 0.00	15.00 ± 0.00	*30*
	S2	4	4	100	9.38 ± 0.00	13.00 ± 0.00	*30*
	S3	10	10	100	17.00 ± 0.00	37.00 ± 0.00	*30*
MRPF	S1	2	2	100	5.50 ± 0.00	11.00 ± 0.00	*30*
	S2	4	4	100	13.38 ± 0.00	21.50 ± 0.00	*30*
	S3	10	10	100	33.30 ± 0.00	56.00 ± 0.00	*30*

Notes. AvgT and Makespan are reported as mean $\pm$ SD across repeated runs of each configuration. The **n** column denotes the number of independent runs used to compute the statistics for that row;

This benchmark evaluation adopts a strict dual-seed management mechanism to ensure fair cross-algorithm comparisons and to scientifically quantify the inherent performance variability of stochastic methods. Specifically, a fixed global environment seed (ENV_SEED = 42) was used to generate identical warehouse layouts, obstacle configurations, and task instances, ensuring that all algorithms faced the same static problem background. Meanwhile, for each independent run, a unique execution seed (EXEC_BASE + k)was assigned to drive the internal random decision-making processes of the algorithms. This separation between environment generation from algorithm execution represents a standard and rigorous practice in the evaluation of randomized optimization algorithms. It also explains why the statistical results obtained from batch experiments differ from the initial single-run outcomes previously reported, as the inclusion of multiple execution seeds naturally captures performance fluctuations.

From the Table 4.31, it can be observed that only the ACO+BA algorithm shows non-zero standard deviation in its performance indicators, whereas the standard deviations of the other three algorithms are all 0.00. This phenomenon is not an error. As a meta-heuristic algorithm, ACO+BA relies on stochastic processes to explore the complex solution space. Therefore, its

performance fluctuates within a certain range under different execution seeds, which is an inherent characteristic of it. In contrast, BFTC, CBPA, and MRPF in the current experimental configuration have deterministic decision-making logic; when faced with the same fixed problem instance, they always converge to exactly the same result, thus demonstrating extremely high stability and predictability.

The experiment further highlights clear performance differences among the algorithms under varying workload pressures. First, regarding task completion rate (Rate), ACO+BA, CBPA, and MRPF achieved perfect performance across all scenarios, successfully completing all assigned tasks.

In contrast, BFTC exhibited consistently low task completion rates in both low-load and high-load settings. The fundamental reason for this lies in its blockchain-based high-latency consensus mechanism. As analyzed in Chapter 3, every state update in the BFTC system requires triggering the Byzantine fault tolerance consensus of all nodes through the DeliverTX message. This is an extremely costly “write operation". Additionally, system time progression depends on an auxiliary role that must wait for all robots to submit transactions before execution can proceed. In high-throughput warehouse environments, increasing numbers of tasks and robots significantly amplify the volume of transactions requiring consensus. Consequently, robots are unable to complete the planning, submission, and consensus processes within the simulation time horizon, leading to widespread task timeouts or abandonment. These findings indicate that although BFTC provides advantages in security, its performance overhead renders it unsuitable for real-time, efficiency-critical warehouse logistics applications.

With respect to efficiency metrics, MRPF achieved the best under the low-load scenario (S1), producing the lowest average completion time (AvgT) and makespan. However, as task volume increased, MRPF’s scalability limitations became evident. In the high-load scenario (S3), its makespan rose sharply to 56.00, making it the least efficient among all evaluated algorithms. By comparison, both ACO+BA and CBPA demonstrated more balanced and scalable behavior. Even under high-load conditions, their completion times remained within reasonable bounds, outperforming MRPF and BFTC in terms of overall efficiency and robustness.

These experimental findings directly address the research questions raised in Chapter 3. Both MRPF and BFTC reveal significant weaknesses in modern warehouse environments. Although MRPF’s centralized architecture enables rapid response under light workloads, its inherent single-point bottleneck leads to poor scalability under heavy loads. Similarly, while BFTC’s rigorous consensus mechanism provides resilience against Byzantine failures [22], it introduces excessive overhead that severely reduces task success rates. On the contrary, FT-CBPA and FT-ACO+BA, which represent the integrated fault-tolerant architecture, successfully achieved a balance between efficiency and robustness by attaching the modular recovery protocol (R3P) to an effective distributed scheduling core. They not only guarantee a 100% task completion rate in all scenarios, but also exhibit excellent scalability and efficiency.

Therefore, the experimental results strongly support the conclusion that for high-throughput, large-scale warehouse applications, integrated fault-tolerant architectures are superior to the native fault-tolerant architecture. This paradigm allows practitioners to first select the most efficient scheduling core under fault-free conditions and subsequently enhance its fault tolerance, offering a more flexible and practical design strategy.

In the overall comparison, FT-CBPA demonstrated the most desirable performance. Unlike the centralized bottleneck of MRPF, CBPA avoids centralized bottlenecks through distributed consensus, providing strong scalability for large robot fleets. Furthermore, compared with FT-ACO+BA, CBPA explicitly incorporates realistic robot workload capacity into its decision-making, making it more aligned with practical warehouse constraints. Based on these experimental outcomes and theoretical analysis, CBPA has been selected as the foundation for further research and optimization in Chapter 5.

Chapter 5

Proposed Framework: SPA-CBPA Algorithm based on Successor Pre-allocation

This chapter evaluates our proposed approach against the strongest integrated baseline identified in the above chapters. All results are reported relative to FT-CBPA, using identical protocols and map configurations to ensure a fair comparison. We depart from prediction-based recovery and instead adopt a preparatory–proactive routing strategy. Rather than reacting after failures occur, the method embeds a contingency takeover plan directly into the consensus process.

To this end, we introduce SPA-CBPA, a successor pre-allocation overlay for CBPA that incorporates a lightweight contingency plan during consensus. By design, SPA-CBPA reduces recovery latency and communication overhead compared with reactive re-auctioning, while fully preserving the core semantics and guarantees of CBPA.

5.1 Analysis of the Passive Re-auctioning Mechanism

Although FT-CBPA improves the robustness of the basic CBPA by introducing a Reactive Re-auctioning Protocol, its fault-tolerance strategy remains fundamental post-event and remedial in nature. As a result, it exhibits three inherent and difficult-to-mitigate limitations [10] when deployed in high-intensity real-world scenarios.

Firstly, FT-CBPA suffers from high recovery latency[12], a critical bottleneck during the global re-planning process. Once a robot failure is detected, the system initiates a global re-planning process that involves multiple rounds of communication. During this interval, all healthy robots suspend their ongoing tasks to participate in re-bidding, negotiation, and consensus formation for the reassignment of " orphan tasks" [19]. This temporary decision vacuum directly degrades system performance, leading to increased Makespan and reduced Throughput.

Second, the recovery process incurs significant resource overhead. Each re-auction requires all healthy robots to recompute their task costs, then broadcast the bidding information and engage in the next round of consensus. In distributed robotic systems where communication bandwidth and computational resources are inherently constrained, such repeated coordination constitutes a substantial inefficiency. In an environment with frequent failures, frequently allocating resources to internal coordination rather than performing core tasks will weaken the overall efficiency of the system [23].

Finally, FT-CBPA exhibits a non-deterministic recovery trajectory. In emergency situations, the outcome of the re-auction process is inherently unpredictable, and the system cannot guarantee that orphan tasks will be reassigned to the most suitable robots. This uncertainty increases the risk of suboptimal task allocations and may trigger cascading failures [10] across the system.

Collectively, these limitations demonstrate that the passive recovery mechanism inherently sacrifices the speed, efficiency, and stability of the recovery process. This observation motivates the exploration of a new fault-tolerance paradigm: transitioning from reactive recovery to proactive fault-aware planning that is embedded directly into the task allocation process. This paradigm shift[42] forms the core motivation for the SPA-CBPA framework proposed in this chapter [15].

5.2 SPA-CBPA Framework: Consensus Algorithm based on Successor Pre-allocation

We now formalize the SPA-CBPA framework and present the corresponding pseudocode for successor selection and task takeover. The background motivation and theoretical justifications have been presented in the preceding chapters.

To address these challenges, this chapter proposes a consensus-based load allocation algorithm with successor pre-allocation (SPA-CBPA). The proposed mechanism is seamlessly embedded into the standard CBPA consensus protocol [36] to enhance fault-tolerance. Specifically, fault-tolerance is enhanced by extending the original consensus objective, which is from agreeing on a single task “winner" to jointly agreeing on a (winner, successor) binary pair for each task. This design enables rapid and deterministic recovery in multi-robot systems by allowing pre-designated successors to immediately take over tasks in the event of failures.

5.2.1 Problem Formalization

Let R denote the robot index set and T denote the task index set. Each robot $i \in \mathrm{R}$ maintains an ordered route π_i. When task $j \in \mathrm{T}$ is committed to winner $w_j \in \mathrm{R}$, we additionally designate a *successor* $s_j \in \mathrm{R} \setminus \{w_j\}$ and record a binary successor matrix $Z \in \{0,1\}^{|\mathrm{T}|\times|\mathrm{R}|}$ with $Z_{js_j} = 1$ (and 0 otherwise). The successor remains dormant unless w_j fails before starting task j.

5.2.2 Extended Consensus and Successor Selection

This section presents the core innovation of SPA-CBPA, which extends the consensus protocol to allow simultaneous agreement on both a primary executor and an emergency executor for each task.

In the first stage, consensus is reached on the primary executor (winner). This stage is identical to the standard CBBA/CBPA protocol. Robot agents broadcast their bids (i.e., marginal benefits) for each task, and through iterative local communication, the network converges to an agreement on the highest bidder (equivalently, the lowest-cost agent) for each task. The selected agent is then designated as the task winner [26].

Once the winner of a task is determined, the successor is chosen by minimizing a prospective takeover cost:

$$s_j \in \arg\min_{s \neq w_j} \Delta C_s(j \mid \pi_s) \; + \; \lambda \, \Phi_s(\pi_s) \; + \; \mu \, \mathbb{I}[\text{local conflict risk}], \tag{5.1}$$

where $\Delta C_s(\cdot)$ is the baseline insertion cost; $\Phi_s(\pi_s)$ penalizes overloaded/long routes; $\mathbb{I}[\cdot]$ is a risk flag for chokepoints. Hyperparameters $\lambda, \mu \geq 0$ are soft weights.

In this work, we consider two approaches: (i) *Static* successor, where we set $\lambda = \mu = 0$, simplifying the choice to the runner-up bidder. This is the primary implementation for our experiments. (ii) *Dynamic* successor, where $\lambda, \mu > 0$, which represents a more advanced selection strategy and a direction for future work.

The SPA-CBPA process and pseudocode are presented in the following:

5.2.3 SPA-CBPA Process and Pseudo-code

The SPA-CBPA process and pseudocode are presented in the following:

Algorithm 5.1: SPA-CBPA Part 1: Successor Pre-allocation During Consensus

Function DETERMINESUCCESSORS($\mathbf{X}_i, \mathbf{T}_i, \mathbf{L}_i, \mathit{LocalBids}$)
  foreach *task* $j \in \mathscr{J}$ **do**
    $w_j \leftarrow \arg\min_{k \in \mathscr{I}} \{\texttt{LocalBids}[k][j]\}$ `// Winner: lowest bid`
    `CandidateBids` $\leftarrow$ `LocalBids` $\setminus \{w_j\}$;
    if $|\mathit{CandidateBids}| > 0$ **then**
      $s_j \leftarrow \arg\min_{k \in \texttt{CandidateBids}} \{\texttt{LocalBids}[k][j]\}$ `// Successor`
      $\mathbf{Z}_i[j][s_j] \leftarrow 1$ `// Mark successor in matrix`
    else
      $\mathbf{Z}_i[j][\cdot] \leftarrow 0$ `// No successor available`
    end
    $\mathbf{X}_i[j][w_j] \leftarrow 1$ `// Mark winner`
  end
  return $\mathbf{X}_i, \mathbf{Z}_i$;
end

Algorithm 5.1 illustrates the core innovation of the SPA-CBPA framework, namely the pre-allocating of successors during the consensus stage. For each task j, the algorithm performs a dual decision-making process. First, following standard auction principles, the task is assigned to the robot with the lowest bid, which is designated as the winner w_j. Next, among the remaining bidders, the robot with the second-lowest bid is selected as the successor s_j for the task. The resulting (w_j, s_j) pairing is recorded in the updated winner matrix $\mathbf{X}_i$ and the newly introduced successor matrix $\mathbf{Z}_i$, thereby embedding a contingency mechanism directly into the initial task allocation process.

Algorithm 5.2: SPA-CBPA Part 2: Failure Detection and Successor Activation

```
1  Function HANDLEFAILUREEVENT(msg, X_i, Z_i, L_i, b_i, p_i)
2      if msg.type = FAULT_ANNOUNCEMENT then
3          w_fail ← msg.failed_robot;
4          foreach task j ∈ 𝒥 do
5              if X_i[j][w_fail] = 1 then
6                  ACTIVATESUCCESSOR(j, w_fail) // Failed robot was winner
7              end
8          end
9      else
10         if msg.type = TASK_TAKEOVER then
11             SYNCHRONIZEGLOBALSTATE(msg);
12         end
13     end
14 end
15 ;
16 Function ACTIVATESUCCESSOR(j, w_fail)
17     s ← FINDSUCCESSOR(Z_i, j) // Get pre-allocated successor
18     if s = i and ¬IS_FAILED(i) then
           // Level 1:  I am a successor
19         X_i[j][w_fail] ← 0; X_i[j][i] ← 1;
20         pos ← FINDOPTIMALINSERTPOSITION(j, p_i);
21         b_i.insert(j); p_i.insert(j, pos);
22         if CHECKPAYLOADCONSTRAINTS(i, j, L_i) then
23             L_i ← UPDATEPAYLOADALLOCATION(i, j, L_i);
24             τ_j ← UPDATETASKSTARTTIME(j, X_i, T_i);
25             broadcast TASK_TAKEOVER(i, j);
26         else
27             INITIATELEVEL2RECOVERY(j) // Insufficient payload
28         end
29     else
30         if s ≠ NULL and IS_FAILED(s) then
31             INITIATELEVEL2RECOVERY(j) // Successor also failed
32         end
33     end
34 end
```

Algorithm 5.2 describes the second stage of the SPA-CBPA framework, which focuses on fault detection and the successor activation mechanism. The algorithm is event-driven. When the core function HANDLEFAILUREEVENT receives a FAULT_ANNOUNCEMENT

message, it identifies all “orphan tasks" previously assigned to the failed robot and invokes the ACTIVATESUCCESSOR function for each of these tasks to initiate the recovery process [21].

The ACTIVATESUCCESSOR function is the key to the protocol, as it implements a hierarchical recovery strategy. The algorithm first searches the successor matrix $\mathbf{Z}_i$ to locate the pre-assigned successor s_j for task j. If the current robot corresponds to this successor and is in a healthy operational state, the task is inserted into its route, and the local state is updated accordingly. The robot then broadcasts a TASK_TAKEOVER message to notify the network of the successful task reassignment. In addition, the algorithm incorporates a robust fallback mechanism. If the designated successor has also failed or is unable to assume the task due to capacity constraints, the protocol triggers the LEVEL2RECOVERY mechanism, which initiates a local re-auction. This design ensures resilient system recovery even under multiple or cascading failure scenarios.

Algorithm 5.3: SPA-CBPA Part 3: Level 2 Recovery and State Synchronization

Function INITIATELEVEL2RECOVERY(j)
  best_robot $\leftarrow$ NULL; best_cost $\leftarrow \infty$;
  foreach *healthy robot* $k \in \mathscr{I}$ **do**
    $c_k \leftarrow$ CALCULATEINSERTIONCOST(k, j);
    if $c_k <$ *best_cost* ***and*** MEETSPAYLOADREQ(k, j) **then**
      best_robot $\leftarrow k$; best_cost $\leftarrow c_k$;
    end
  end
  if *best_robot* $\neq$ *NULL* **then**
    if *best_robot* $= i$ **then**
      ASSIGNTASKTOSELF(j);
    end
    broadcast LEVEL2_ASSIGNMENT(best_robot, j);
  else
    INITIATEGLOBALRE-AUCTION(j) // Fallback to global re-auction
  end
end
;
Function SYNCHRONIZEGLOBALSTATE(*msg*)
  $j \leftarrow$ msg.task;
  $s_{\text{new}} \leftarrow$ msg.new_owner;
  $w_{\text{old}} \leftarrow$ FINDCURRENTWINNER($j, \mathbf{X}_i$);
  if $w_{old} \neq$ *NULL* **then**
    $\mathbf{X}_i[j][w_{\text{old}}] \leftarrow 0$;
  end
  $\mathbf{X}_i[j][s_{\text{new}}] \leftarrow 1$;
  $\mathbf{T}_i[j][s_{\text{new}}] \leftarrow$ ESTIMATEARRIVALTIME(s_{new}, j);
  $\tau_j \leftarrow \max_{k:\mathbf{X}_i[j][k]=1}\{\mathbf{T}_i[j][k]\}$ // Update start time
end

Algorithm 5.3 details the final component of the SPA-CBPA framework, which addresses secondary recovery and global state synchronization. This stage is essential for enabling the system's downgrade capability when primary recovery mechanisms are unsuccessful.

The INITIATELEVEL2RECOVERY function is activated when first-level recovery fails. It performs a local re-auction among all healthy robots by iterating through candidate agents, computing and comparing their costs to assume the orphan task j, and verifying their remaining load capacity. The robot that satisfies all feasibility constraints and yields the lowest cost is selected as the new executor, after which a LEVEL2_ASSIGNMENT message is broadcast to inform the network. If no suitable candidate is identified during the local re-auction, the

function escalates the process by invoking INITIATEGLOBALREAUCTION to initiate a global re-auction.

The SYNCHRONIZEGLOBALSTATE function is responsible for maintaining the consistency of the distributed system. Whenever any robot receives a message indicating a change in task ownership (whether through primary or secondary recovery), this function updates its local winner matrix $\mathbf{X}_i$ and time matrix $\mathbf{T}_i$, ensuring that all robots have a unified and conflict-free view of the current task allocation scheme.

5.2.4 Theoretical Comparison of SPA-CBPA and FT-CBPA

Table 5.1 Comparison of FT-CBPA and SPA-CBPA Mechanisms

Metric	**FT-CBPA**	**SPA-CBPA**	**Advantage**
Trigger	Reactive	Proactive	Faster detection
Recovery	Global re-auction	Local activation	Decentralized
Communication	Multi-round	$O(1)$	Speed improvement
Message cost	$O(N^2)$	$O(1)$	Load reduction
Computation	High ($N-1$ robots)	Low (1 robot)	Resource efficiency
Recovery path	Non-deterministic	Deterministic	Predictability

To clearly quantify and compare the differences in resource consumption and performance characteristics between the two recovery protocols, Table 5.1 presents a systematic comparison of the passive re-auction mechanism used in FT-CBPA and the active successor activation mechanism proposed in SPA-CBPA. The comparison is conducted across six key dimensions.

The table highlights fundamental differences in the core mechanisms of the two approaches. Although both protocols are triggered by the same fault-detection event, FT-CBPA relies on a global re-auction and consensus process. This process requires multiple rounds of communication to complete the bidding and conflict resolution, leading to high message complexity and computational overhead that scale with the number of healthy robots $(N-1)$ and the remaining tasks (T_r).

By contrast, SPA-CBPA utilizes a local successor activation mechanism that reduces the recovery communication to a constant number of broadcasts (two in total), thereby significantly reducing both message complexity and computational load. Replanning is required only for the designated successor robots, while the rest of the network remains unaffected. Moreover, because FT-CBPA conducts real-time auctions under failure conditions, its recovery paths are inherently non-deterministic. In contrast, SPA-CBPA achieves deterministic recovery behavior through precomputed successor assignments.

In summary, Table 5.1 provides a theoretical comparison demonstrating the advantages of SPA-CBPA over FT-CBPA in terms of recovery latency, network and computational efficiency, and predictability. These insights establish a strong theoretical basis for the empirical evaluations presented in the subsequent sections.

5.3 Experimental Design and Experimental Verification

5.3.1 Simulation Setup and Baseline Models

We reuse the two-dimensional warehouse simulator of Chapte 4 (10×10 grid, fixed shelf obstacles, identical start corners, shared task pools, fixed random seeds). Two recovery strategies are evaluated:

- **FT-CBPA (reactive re-auction).** A baseline that triggers a local/global re-auction when a winner fails; no pre-designated successors.
- **SPA-CBPA (successor overlay).** Our method augments CBPA with a pre-designated successor for each task and a constant-size takeover protocol. We report both *Static* (runner-up successor) and *Dynamic* (penalized scoring) variants.

Robots R=4 and tasks $T \in \{10,20,30,40\}$ share the same map and random-seed pool. For each $(\text{algorithm}, T)$, we run N=30 independent trials and report **mean±std**. We evaluate two strategies: **FT-CBPA** (reactive re-auction, code label: "`Re-auction`") and **SPA-CBPA** (successor overlay, code label: "`SPA-Enhanced`"). Static/Dynamic variants of successors are kept as methodological context, but all tables in this chapter use the **current outputs** only.

Significance is assessed by paired tests across N=30 runs per (algorithm, T); we report p-values alongside mean±std in tables.

5.3.2 Evaluation Metrics

Let F be the set of failure events that generate orphan tasks. For an orphan task j, the recovery latency is

$$L_{\text{rec}}(j) = t_{\text{commit}}(j) - t_{\text{detect}}(j), \tag{5.2}$$

where $t_{\text{detect}}(j)$ is the timestamp when the winner's failure affecting j is detected, and $t_{\text{commit}}(j)$ is the timestamp when the successor takeover (or the re-auction assignment) is acknowledged.

The recovery message overhead for j is the number of protocol messages exchanged *solely* for recovering j, denoted $M_{\text{rec}}(j)$. For SPA-CBPA Level-1 takeover is near-constant (one or two broadcasts), while for FT-CBPA re-auction, it scales with the number of healthy robots.

We report $\overline{L}_{\text{rec}} = \frac{1}{|\text{F}|}\sum_{j\in\text{F}} L_{\text{rec}}(j)$ and $\overline{M}_{\text{rec}} = \frac{1}{|\text{F}|}\sum_{j\in\text{F}} M_{\text{rec}}(j)$, together with system-level side-effects: makespan, completion rate, and Level-1/Level-2 recovery counts. In addition,

we report **mean±standard deviation** over N=30 independent runs per $(\text{algorithm}, T)$, consistent with our current implementation and outputs.

Table 5.2 Primary metrics and expected trends.

Metric	Definition	Trend (SPA)
Average Recovery latency ($\overline{L}_{\text{rec}}$)	$t_{\text{commit}} - t_{\text{detect}}$ averaged over failure events	Lower
Average Message overhead ($\overline{M}_{\text{rec}}$)	Messages used only for recovery (successor takeover or re-auction)	Reduced
Makespan / Conflicts	System-level side-effects	Comparable

All metrics follow the common definitions of Chapter 3 to ensure strict comparability.

5.3.3 Fault Triggering Mechanisms

Legacy iteration-coupled trigger (reference only)

During early debugging, we used a deterministic, iteration-coupled trigger: at a fixed planning iteration, the first eligible robot is forced to fail, and its first task is marked completed; all remaining tasks on its route become orphans. This mechanism is fully reproducible but weakly correlated with physical-time execution.

Time-based random-failure injector (used in this chapter)

We design a time-based injector that is decoupled from the planning loop and operates on continuous execution time.

Assumption 1 (Open-interval sampling)**.** *For a failing robot $i \in \mathrm{R}$ with planned route π_i, the total execution time estimate is $\widehat{T}_i(\pi_i) > 0$. The failure time is sampled from an open interval*

$$t_{\text{fail}} \sim U\big(\varepsilon,\ \widehat{T}_i(\pi_i) - \varepsilon\big), \qquad \textit{with } 0 < \varepsilon \ll \widehat{T}_i(\pi_i),$$

thus avoiding edge cases at the start/end of the route.

Assumption 2 (Phase mapping)**.** *Let $\psi_i(t) \in \{\text{TRANSIT}, \text{TASK}, \text{AFTER}\}$ denote the execution phase of robot i at time t, obtained by integrating its kinematic model and service times along π_i. The mapping $\psi_i(\cdot)$ is computed independently of the planner's iteration counter.*

Assumption 3 (Completion-upon-arrival rule)**.** *Let j be the current task of robot i at t_{fail}. If the robot has reached the task location before t_{fail}, then j is counted as* completed *regardless of how much service time remains; otherwise j becomes an orphan. All tasks not yet started after j are orphans. This rule separates logical planning from continuous-time execution.*

Formal definition. Let $O_i(t_{\text{fail}}) \subseteq \mathrm{T}$ be the set of orphan tasks produced by the Failure of robot i. With Assumptions 1–3,

$$O_i(t_{\text{fail}}) = \begin{cases} \{j\} \cup \{j' \in \pi_i : j' \text{ scheduled after } j\}, & \text{if } \psi_i(t_{\text{fail}}) \in \{\text{TRANSIT}, \text{TASK}\} \\ & \text{and } j \text{ not reached}, \\ \{j' \in \pi_i : j' \text{ scheduled after current task } j\}, & \text{otherwise.} \end{cases} \tag{5.3}$$

Algorithmic procedure (EnhancedFaultInjector).

1. Select a failing robot i according to the experiment configuration (uniform among robots with nonempty π_i unless specified).
2. Compute $\widehat{T}_i(\pi_i)$ by summing travel times (from the kinematic model) and service times along π_i.
3. Sample $t_{\text{fail}} \sim U\left(\varepsilon, \widehat{T}_i(\pi_i) - \varepsilon\right)$.
4. Determine $\psi_i(t_{\text{fail}})$ and the current task j by integrating the timeline; compute the physical pose $q_i(t_{\text{fail}})$.
5. Apply Assumption 3 to decide whether j is completed or orphaned; form $O_i(t_{\text{fail}})$ by appending all not-yet-started tasks.
6. Emit the failure event $\langle i, t_{\text{fail}}, \psi_i(t_{\text{fail}}), O_i(t_{\text{fail}}) \rangle$ to the recovery module (SPA-CBPA or FT-CBPA).

5.3.4 Comparative Results across Load Regimes

Table 5.3 Multi-scale comparison over N=30 runs (mean±std). Δ vs T10: increase relative to the same algorithm's T=10; Δ vs Baseline: increase relative to the other algorithm at the same T. Latency unit: seconds.

T	Algorithm	CR (%)	Makespan	Δ vs T10 (%)	Δ vs Baseline (%)	L_{rec} (s)	Comm. Overhead	L1 / L2
10	SPA-CBPA	100.00±0.00	35.03±6.65	0.00	38.47	0.00±0.00	2.33±1.12	2.33±1.12 / 0.00±0.00
10	FT-CBPA	100.00±0.00	25.30±2.00	0.00	-27.78	0.00±0.00	5.80±2.48	0.00±0.00 / 1.93±0.83
20	SPA-CBPA	100.00±0.00	51.43±4.91	46.81	19.84	0.00±0.00	3.47±1.68	3.47±1.68 / 0.00±0.00
20	FT-CBPA	100.00±0.00	42.92±2.98	69.63	-16.56	0.02±0.01	8.70±4.12	0.00±0.00 / 2.90±1.37
30	SPA-CBPA	100.00±0.00	73.60±10.20	110.09	26.10	0.00±0.00	4.63±2.44	4.63±2.44 / 0.00±0.00
30	FT-CBPA	100.00±0.00	58.37±3.83	130.70	-20.70	0.04±0.02	12.50±6.83	0.00±0.00 / 4.17±2.28
40	SPA-CBPA	100.00±0.00	91.23±10.36	160.42	21.92	0.00±0.00	5.70±2.97	5.70±2.97 / 0.00±0.00
40	FT-CBPA	100.00±0.00	74.83±6.37	195.78	-17.98	0.07±0.04	16.40±8.33	0.00±0.00 / 5.47±2.78

Under this unified protocol, both methods achieve 100% completion across all loads and makespan increases with T. At light load (T=10) FT-CBPA yields a lower average makespan than SPA–CBPA, while at moderate-to-heavy loads ($T \geq 20$) the gap narrows although FT-CBPA remains generally faster on average. The mechanism-level picture is consistent with

design: SPA–CBPA relies on a pre-designated successor for deterministic Level-1 takeover, producing near-constant L_{rec} and constant-size communication, whereas FT-CBPA reduces makespan via participatory re-auction at the cost of recovery traffic that scales with the number of healthy robots. The Δ *vs T10* and Δ *vs Baseline* columns quantify, respectively, within-method expansion with load and cross-method differences at each T.

5.3.5 Successor policy Ablation inside SPA–CBPA

We conduct an ablation study to address a focused but important question: once a successor overlay is introduced (deterministic Level-1 takeover), does the specific heuristic used to select successors meaningfully affect recovery performance? This isolates the role of the mechanism (having a successor overlay at all) from the heuristic (details of who the successor is). If recovery performance is primarily driven by the mechanism, both heuristics should produce nearly identical latency and communication overhead profiles. In contrast, consistent and significant differences would suggest that the heuristic plays the dominant role.

Table 5.4 evaluates two successor policies under the same environment, task pools, maps, and seed set as the main study: *Static* (choosing the runner-up of the initial assignment as the successor) and *Dynamic*(online penalized scoring). We vary nothing else in the pipeline—initial allocation, time-based failure injector, metric definitions, and reporting protocol are identical. For each load $T \in \{10, 20, 30, 40\}$ we execute N=30 independent trials and report mean$\pm$std for makespan, recovery latency $\overline{L}_{\text{rec}}$, recovery messages (*Comm. Overhead*), and the Level-1/Level-2 recovery counts.

Table 5.4 Ablation on successor policy within SPA–CBPA over N=30 runs (*mean*$\pm$*std*). All settings achieve CR= 100%; we therefore omit CR from the table to save space. Latency unit: seconds.

T	Variant	Makespan	L_{rec} (s)	Comm. Overhead	Level-1	Level-2
10	SPA-Static	35.22 ± 4.82	0.00 ± 0.00	2.13 ± 1.01	2.13 ± 1.01	0.00 ± 0.00
10	SPA-Dynamic	34.45 ± 5.06	0.00 ± 0.00	2.13 ± 1.07	2.13 ± 1.07	0.00 ± 0.00
20	SPA-Static	51.05 ± 5.86	0.00 ± 0.00	3.31 ± 1.70	3.31 ± 1.70	0.00 ± 0.00
20	SPA-Dynamic	51.01 ± 5.13	0.00 ± 0.00	3.13 ± 1.70	3.13 ± 1.70	0.00 ± 0.00
30	SPA-Static	76.68 ± 10.77	0.00 ± 0.00	5.33 ± 2.53	5.33 ± 2.53	0.00 ± 0.00
30	SPA-Dynamic	72.48 ± 8.27	0.00 ± 0.00	5.33 ± 2.53	5.33 ± 2.53	0.00 ± 0.00
40	SPA-Static	90.85 ± 10.75	0.00 ± 0.00	5.50 ± 2.71	5.50 ± 2.71	0.00 ± 0.00
40	SPA-Dynamic	88.12 ± 10.25	0.00 ± 0.00	5.33 ± 2.99	5.33 ± 2.99	0.00 ± 0.00

Across all load levels, the two policies demonstrate almost identical recovery behavior. Recovery latency remains at (near) zero for both policies, and communication overheads are effectively the same, increasing with T in step with the number of Level-1 takeovers, while Level-2 fallbacks are virtually absent. This pattern reflects the characteristic signature of the successor overlay mechanism: deterministic Level-1 takeover completes within a constant number of protocol steps. As a result, both latency and message complexity are determined by the existence of the overlay itself, rather than by the specific choice of successor robot.

Differences in makespan are minor and load-dependent (e.g., Dynamic is modestly lower at T=30 and T=40), likely due to scenario-specific route reshaping or temporary congestion mitigation. However, these variations do not alter the overall recovery dynamics.

Under our experimental settings, **the mechanism > is heuristic**. Adopting a successor overlay determines the recovery characteristics (near-constant$\overline{L}_{\text{rec}}$, constant-size communication, Level-1 dominance), while the particular successor heuristic (Static vs. Dynamic) provides only marginal variations. Practically, SPA–CBPA remains a robust default for predictable low-overhead recovery; either policy can be used without materially affecting $\overline{L}_{\text{rec}}$, communication, or the Level-1/Level-2 mix.

5.3.6 Experimental Results Summary and Discussion

The comprehensive experimental results in this section systematically validate both the practical value and internal mechanisms of the proposed SPA-CBPA framework from two distinct levels.

First, the cross-load comparisons reported in **Table 5.3** clearly quantify the fundamental trade-off between "proactive preparatory" and "passive reactive" fault tolerance paradigms. On one hand, SPA-CBPA consistently achieves near-zero recovery latency ($L_{rec} \approx 0.00s$) and significantly lower communication overhead across all task loads (e.g., at T=40, the overhead is merely 34.8% of that of FT-CBPA). These results clearly demonstrate the decisive advantage of the successor pre-allocation mechanism in enabling fast, lightweight, and deterministic recovery. On the other hand, the data also make explicit the cost of this proactive design: SPA-CBPA increases of approximately 20% to 38% in the system's overall makespan compared to FT-CBPA. As discussed in Section 5.3.4, this quantitative result exposes the "opportunity cost" paid by SPA-CBPA to gain its superior recovery performance. The system sacrifices a portion of its global execution efficiency to ensure instantaneous recovery.

Second, the ablation study in **Table 5.4** further clarifies the key drivers of the SPA-CBPA framework's success. By comparing the "Static" and "Dynamic" successor selection policies, the results show that both strategies yield nearly identical performance in core recovery metrics (recovery latency L_{rec}, communication overhead, and L1/L2 recovery counts). This finding is highly significant as it strongly supports the central dissertation presented in Section 5.3.5: **"the mechanism > is heuristic"**. In other words, the performance gains of SPA-CBPA are primarily derived from the *existence of the successor pre-allocation mechanism itself*, rather than the specific heuristic rule used to select which robot becomes the successor. Although a more complex dynamic policy can lead to marginal improvements in makespan under high loads, it does not alter the fundamental recovery behavior.

In summary, the experimental evidence demonstrates that SPA-CBPA successfully transforms fault recovery from a slow, communication-intensive, and uncertain passive process into a deterministic, lightweight, and near-instantaneous "proactive plan activation" process.

Despite this coming at the cost of some routine operational efficiency, the order-of-magnitude improvement in fault response speed and predictability, combined with the robust nature of its core advantages being independent of complex heuristic rules, establishes it as an exceptionally valuable solution for application scenarios with stringent requirements for system reliability and task continuity.

5.4 Discussion

5.4.1 Algorithm Empirical Testing and Baseline Establishment

Table 5.5 Comparison of Algorithm Validation and Baseline Establishment

Algorithm Category	Representative Algorithm	Core Mechanism	Experimental Characteristics	Conclusion
Native-Centralized	MRPF	Optimal preemption + nearest neighbor exchange	Significant performance degradation under S1 scenario (Avg$\tau = 5.50s$), but stable improvement in task completion under S3 load (Makespan $= 56.00s$), susceptible to central node bottleneck.	Not suitable for large-scale scenarios
Native-Distributed	BFTC	Blockchain consensus + token economy	Under all load scenarios (S3(42.40%)), consensus cost increases significantly with cluster size, not suitable for high concurrency flow scenarios.	Relatively low efficiency
Integrated-Heuristic	FT-ACO+BA	Data grouping + ant colony algorithm co-optimization	Achieved 100% task completion, but self-recovery capability of random task addition is weak (Std Dev > 0), and the algorithm time is longer.	Relatively stable
Integrated-Field Method	FT-CBPA	Load-aware positioning + potential field repulsion	Excellent performance, 100% task completion in all load scenarios, S3 load τ_{Makespan} (37.00s) approaches MRPF, demonstrating excellent robustness.	Relatively optimal

To comprehensively evaluate the performance of the proposed 2D warehouse platform, four representative baseline algorithms were selected under low (S1), medium (S2), and

high (S3) load scenarios. The experiments conducted a full-scale quantitative evaluation of the four representative algorithms, and the experimental results not only validated the effectiveness of the theoretical framework but also further highlighted the advantages of the baseline algorithm.

5.4.2 The Resilience-Efficiency Trade-off

In the comparative experiment presented in Chapter 5, SPA-CBPA achieved zero delay recovery, but its makespan increased by 20% to 38% compared to FT-CBPA. This result explicitly quantifies the cost of proactivity.

These findings suggest that achieving ultimate recovery speed typically requires sacrificing part of the optimality of global planning, precisely because the deterministic pre-allocation in SPA-CBPA bypasses real-time global re-optimization at the moment of failure, thereby trading nominal path-overlapping efficiency for instantaneous resilience. This trade-off defines a resilience-efficiency Pareto frontier in multi-robot systems. In practical engineering applications, an appropriate operating point along this frontier must be selected according to the Service Level Agreement (SLA) requirements of the specific business scenario—namely, whether recovery timeliness or overall execution cost is the primary priority.

5.4.3 Mechanism Determinism

This study derives a theoretically meaningful insight from the ablation experiment. Within the SPA–CBPA framework, we evaluated two fundamentally different successor-selection strategies. The first adopts a simple rule: selecting the robot with the second-lowest bid. The second applies a more sophisticated approach, using dynamic penalty functions and optimization models to compute the “best” successor.

Remarkably, the experimental results reveal almost no performance difference between these two strategies in terms of recovery delay, communication overhead, or even overall makespan. This finding indicates that once the deterministic “pre-allocation" architectural mechanism is introduced, a qualitative change in system resilience has occurred. And the specific micro-selection strategy (heuristic) can only bring marginal improvement. This discovery greatly simplifies the implementation difficulty of active fault-tolerant systems. Developers do not need to design complex scoring models, but they only need to implement a simple suboptimal replacement mechanism to obtain the majority of the resilience benefits.

5.4.4 Reconstruction of Theoretical Framework: The Binary Distinction between Native and Integrated Architectures

During the literature review and theoretical analysis phase, this research critically examines existing fault-tolerance classification systems (such as centralized/distributed, reactive/active) and points out their limitations in guiding the overall design of the system. To address this

gap, we introduce a new binary analytical framework: native fault-tolerant architectures versus integrated fault-tolerant architectures.

A native architecture is represented by the MRPF algorithm and the BFTC algorithm. Its characteristic is that the fault-tolerant logic is embedded into the core scheduling algorithm (Fault Tolerance by Design). This tight coupling, although having extremely strong defense capabilities against specific faults (such as Byzantine attacks), often comes at the expense of normal operational efficiency. For example, the blockchain consensus mechanism introduced by BFTC to maintain global consistency has resulted in unacceptable communication delays and computational costs.

In contrast, an integrated architecture is represented by FT-CBPA and FT-ACO+BA. Its design philosophy is" decoupling. That is, first, the most optimal distributed scheduling kernel (such as CBPA) is selected in a fault-free environment, and then, lightweight fault recovery protocols (such as R3P) are externally attached through a modular approach. This design enables the system to maintain extremely high throughput during normal operation, and only calls for recovery resources when a fault is triggered, demonstrating superior engineering adaptability.

5.5 Summary of Research Innovation Contributions

Table 5.6 Summary of Research Contributions

Contribution Dimension	Specific Content	Academic and Practical Value
Theoretical Innovation	Ablation analysis framework comparing native vs. collective architectures,	Addressed the longstanding controversy over classification homogeneity in MRTA systems. The first study to systematically quantify the advantages of collective architectures through rigorous ablation testing, providing novel theoretical evidence for system design paradigms.
Algorithmic Breakthrough	SPA-CBPA active recovery protocol	Proposed a “preassignment + post-recovery" dual-layer shared mechanism that significantly reduces the computational cost of fault recovery from $O(N)$ to near $O(1)$ by leveraging the determinism inherent in system architecture, thereby resolving the fundamental tension between rapid response and optimization quality in conventional reactive methods.
Validation Infrastructure	Unified evaluation and testing benchmark	Established a comprehensive and standardized testing framework that integrates fault injection capabilities (Fault Injector), diverse baseline algorithms, and holistic performance metrics, thereby providing a systematic benchmark for the research community to evaluate proactive fault-tolerant systems.

Chapter 6

Conclusion and Future Work

This thesis addressed the “resilience crisis" faced by multi-robot systems (MRS) in dynamic warehouse environments. Through a systematic investigation spanning architectural theory to protocol-level reconfiguration, core contributions are validated across theoretical, methodological, and experimental lines. The study introduces a novel binary analytical framework and develops the proactive SPA–CBPA protocol, which successfully collapses the computational complexity of fault recovery from $O(N)$ to near $O(1)$. Experimental evaluations quantitatively confirm that SPA-CBPA eliminates fault recovery delays to nearly zero and reduces recovery-related communication traffic by up to 65.2%, with an acceptable 20% to 38% increase in global makespan. The results demonstrate that advancing recovery decision-making into the task allocation stage substantially improves the system response and operational efficiency under failure conditions.

The key findings of this study are twofold. First, in warehouse logistics scenarios primarily characterized by physical failures, integrated fault-tolerant architectures exhibit stronger engineering adaptability than native designs. Second, the study confirms that a “design-for-failure" paradigm, implemented through a high-performance core combined with plug-in recovery mechanisms, provides the necessary flexibility to handle diverse fault distributions in industrial applications.

Several limitations should be acknowledged. All experimental results were obtained from a discrete-event simulator implemented in Python. Although we integrated the A^* algorithm and the conflict detection mechanism into path planning, simulation cannot fully capture the complexity of real-world physical dynamics. Assumptions such as ideal acceleration/deceleration, perfect gripping force, and the absence of inertia, friction, or terrain irregularities simplify real operational conditions.

Moreover, this study mainly focuses on "Failure to stop responding", where a robot immediately stops responding and disconnects from the network after a failure occurs. This represents a relatively idealized fault model. More complex scenarios, such as partial performance degradation or intermittent communication faults, remain significant open challenges for real-world applications.

Finally, while the proactive mechanism effectively eliminates recovery delay, it inherently introduces a measurable trade-off in total makespan due to the sub-optimal nature of pre-assigned successors compared to global re-optimization.

Future work will focus on several critical directions. Firstly, Multi-Agent Reinforcement Learning (MARL) can be employed to optimize successor selection strategies in large-scale dynamic environments. Secondly, real-time resource status, such as battery levels, vehicle types, and workload states, should be integrated into the pre-allocation protocol to improve capability matching during recovery. Beyond these enhancements, future efforts will transition the proposed protocol from simulation to physical multi-robot platforms to validate its robustness against real-world hardware constraints and sensor noise. Lastly, the recovery logic will be extended to maintain system resilience under complex concurrent multi-fault scenarios, while expanding the path-replanning layers to handle dynamic moving obstacles in shared human-robot workspaces.

In conclusion, this thesis begins with concerns about the vulnerability of intelligent logistics systems and ends with a logically consistent proactive fault-tolerance solution. We demonstrated that through careful design at the architectural level, uncertainties can be internalized as a controllable part of the system. From the integrative approach of FT-CBPA to the proactive breakthrough of SPA-CBPA, this work not only addresses a specific engineering problem but also advances a broader “design for failure" thinking paradigm, contributing toward the development of reliable, autonomous, and sustainable intelligent multi-robot systems.

References

[1] Í. R. da Costa Barros and T. P. Nascimento, "Robotic mobile fulfillment systems: A survey on recent developments and research opportunities," *Robotics and Autonomous Systems*, vol. 137, p. 103729, 2021.

[2] B. P. Gerkey and M. J. Matarić, "A formal analysis and taxonomy of task allocation in multi-robot systems," *The International Journal of Robotics Research*, vol. 23, no. 9, pp. 939–954, 2004.

[3] M. A. Haider, D. Ding, R. Kabir, and Y. Watanobe, "Fault-tolerant framework for dynamic task reassignment in multi-robot systems," *Engineering Proceedings*, vol. 120, no. 1, p. 22, 2026.

[4] M. Soori, R. Dastres, B. Arezoo, and F. K. G. Jough, "Intelligent robotic systems in industry 4.0: A review," *Journal of Advanced Manufacturing Science and Technology*, pp. 2 024 007–0, 2024.

[5] S. Oğuz, E. Garone, M. Dorigo, and M. K. Heinrich, "Proactive-reactive detection and mitigation of intermittent faults in robot swarms," *arXiv preprint arXiv:2509.19246*, 2025.

[6] G. A. Korsah, M. M. B. Dias, and A. Stentz, "A comprehensive taxonomy for multi-robot task allocation," *The International Journal of Robotics Research*, vol. 32, no. 12, pp. 1495–1512, 2013.

[7] S. Oğuz, E. Garone, M. Dorigo, and M. K. Heinrich, "Proactive-reactive detection and mitigation of intermittent faults in robot swarms," *arXiv preprint arXiv:2509.19246*, 2025.

[8] C. Street, B. Lacerda, M. Mühlig, and N. Hawes, "Right place, right time: Proactive multi-robot task allocation under spatiotemporal uncertainty," *Journal of Artificial Intelligence Research*, vol. 79, pp. 137–171, 2024.

[9] A. KA and U. Subramaniam, "A systematic literature review on multi-robot task allocation," *ACM Computing Surveys*, vol. 57, no. 3, pp. 1–28, 2024.

[10] A. Prorok, M. Malencia, L. Carlone, G. S. Sukhatme, B. M. Sadler, and V. Kumar, "Beyond robustness: A taxonomy of approaches towards resilient multi-robot systems," *arXiv preprint arXiv:2109.12343*, 2021.

[11] F. Quinton, C. Grand, and C. Lesire, "Market approaches to the multi-robot task allocation problem: a survey," *Journal of Intelligent & Robotic Systems*, vol. 107, no. 2, p. 29, 2023.

[12] T. Zhang, W. Zhang, and M. M. Gupta, "Resilient robots: Concept, review, and future directions," *Robotics*, vol. 6, no. 4, p. 22, 2017.

[13] D. M. Bossens, S. Ramchurn, and D. Tarapore, "Resilient robot teams: a review integrating decentralised control, change-detection, and learning," *Current Robotics Reports*, vol. 3, no. 3, pp. 85–95, 2022.

[14] J. Gielis, A. Shankar, and A. Prorok, "A critical review of communications in multi-robot systems," *Current robotics reports*, vol. 3, no. 4, pp. 213–225, 2022.

[15] Y. Zhang and J. Jiang, "Bibliographical review on reconfigurable fault-tolerant control systems," *Annual reviews in control*, vol. 32, no. 2, pp. 229–252, 2008.

[16] L. E. Parker and B. Kannan, "Adaptive causal models for fault diagnosis and recovery in multi-robot teams," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2006, pp. 2703–2710.

[17] E. Khalastchi and M. Kalech, "Fault detection and diagnosis in multi-robot systems: A survey," *Sensors*, vol. 19, no. 18, p. 4019, 2019.

[18] M. Wang, P. Zhang, G. Zhang, K. Sun, J. Zhang, and M. Jin, "A resilient scheduling framework for multi-robot multi-station welding flow shop scheduling against robot failures," *Robotics and Computer-Integrated Manufacturing*, vol. 91, p. 102835, 2025.

[19] V. C. Kalempa, L. Piardi, M. Limeira, and A. S. de Oliveira, "Multi-robot preemptive task scheduling with fault recovery: A novel approach to automatic logistics of smart factories," *Sensors*, vol. 21, no. 19, p. 6536, 2021.

[20] ——, "Multi-robot task scheduling for consensus-based fault-resilient intelligent behavior in smart factories," *Machines*, vol. 11, no. 4, p. 431, 2023.

[21] M. B. Dias, R. Zlot, N. Kalra, and A. Stentz, "Market-based multirobot coordination: A survey and analysis," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1257–1270, 2006.

[22] L. Lamport, R. Shostak, and M. Pease, "Byzantine fault tolerance for security in distributed systems," *ACM Trans. on Progr. Languages and Systems*, vol. 4, p. 3, 1982.

[23] K. Strawn and N. Ayanian, "Byzantine fault tolerant consensus for lifelong and online multi-robot pickup and delivery," in *International Symposium Distributed Autonomous Robotic Systems*. Springer, 2021, pp. 31–44.

[24] A. M. Elsayed, M. Elshalakani, S. A. Hammad, and S. A. Maged, "Decentralized fault-tolerant control of multi-mobile robot system addressing lidar sensor faults," *Scientific Reports*, vol. 14, no. 1, p. 25713, 2024.

[25] L. E. Parker, "Alliance: An architecture for fault tolerant multirobot cooperation," *IEEE transactions on robotics and automation*, vol. 14, no. 2, pp. 220–240, 2002.

[26] H.-L. Choi, L. Brunet, and J. P. How, "Consensus-based decentralized auctions for robust task allocation," *IEEE transactions on robotics*, vol. 25, no. 4, pp. 912–926, 2009.

[27] Z. Yan, N. Jouandeau, and A. A. Cherif, "A survey and analysis of multi-robot coordination," *International Journal of Advanced Robotic Systems*, vol. 10, no. 12, p. 399, 2013.

[28] F. Arrichiello, A. Marino, and F. Pierri, "Distributed fault-tolerant control for networked robots in the presence of recoverable/unrecoverable faults and reactive behaviors," *Frontiers in Robotics and AI*, vol. 4, p. 2, 2017.

[29] P. Li, Z. An, S. Abrar, and L. Zhou, "Large language models for multi-robot systems: A survey," *arXiv preprint arXiv:2502.03814*, 2025.

[30] V. Strobel, E. Castelló Ferrer, and M. Dorigo, "Blockchain technology secures robot swarms: A comparison of consensus protocols and their resilience to byzantine robots," *Frontiers in Robotics and AI*, vol. 7, p. 54, 2020.

[31] N. Faci, Z. Guessoum, and O. Marin, "Dimax: a fault-tolerant multi-agent platform," in *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, 2006, pp. 13–20.

[32] S. Islam, R. Lindstrom, and N. Suri, "Dependability driven integration of mixed criticality sw components," in *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*. IEEE, 2006, pp. 11–pp.

[33] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[34] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[35] B. P. Gerkey and M. J. Mataric, "Sold!: Auction methods for multirobot coordination," *IEEE transactions on robotics and automation*, vol. 18, no. 5, pp. 758–768, 2002.

[36] X. Qiu, P. Zhu, Y. Hu, Z. Zeng, and H. Lu, "Consensus-based dynamic task allocation for multi-robot system considering payloads consumption," in *2024 China Automation Congress (CAC)*. IEEE, 2024, pp. 5294–5299.

[37] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE computational intelligence magazine*, vol. 1, no. 4, pp. 28–39, 2007.

[38] X.-S. Yang, "A new metaheuristic bat-inspired algorithm," in *Nature inspired cooperative strategies for optimization (NICSO 2010)*. Springer, 2010, pp. 65–74.

[39] X.-S. Yang and A. Hossein Gandomi, "Bat algorithm: a novel approach for global engineering optimization," *Engineering computations*, vol. 29, no. 5, pp. 464–483, 2012.

[40] F. Zitouni, S. Harous, and R. Maamri, "A distributed solution to the multi-robot task allocation problem using ant colony optimization and bat algorithm," in *Advances in Machine Learning and Computational Intelligence: Proceedings of ICMLCI 2019*. Springer, 2020, pp. 477–490.

[41] F. Duchoň, A. Babinec, M. Kajan, P. Beňo, M. Florek, T. Fico, and L. Jurišica, "Path planning with modified a star algorithm for a mobile robot," *Procedia engineering*, vol. 96, pp. 59–69, 2014.

[42] K. O. Aina, H. Bagheri, and D. I. Goldman, "Fault-tolerant multi-robot coordination with limited sensing within confined environments," in *International Symposium on Distributed Autonomous Robotic Systems*. Springer, 2024, pp. 322–336.

Appendix A

Source Code and Experimental Framework

A.1 Overview and Reproducibility

To ensure the reproducibility of the research presented in this thesis, the complete simulation platform, algorithm implementations, and experimental configurations are provided as open-source software. All core logic and data analysis are organized into six primary Jupyter Notebooks (.ipynb), which encompass the reproduction of baselines, the proposed protocol, and the ablation studies.

A.2 GitHub Repository

The project is conducted on GitHub, providing access to the source code, fixed random-seed configurations, and raw experimental data:

- **Repository URL:** https://github.com/HooD1on/Resilient-MRS-Protocol
- **Environment:** Python 3.10 (Requirements: NumPy, SciPy, Matplotlib)

A.3 Source Code Mapping

The following six notebooks constitute the technical core of the comparative study and the validation of the proposed framework:

1. **MRPF.ipynb**: Implementation of the centralized native fault-tolerant architecture based on preemptive scheduling(Table 4.3).
2. **BFTC.ipynb**: Realization of the distributed native architecture utilizing blockchain-based consensus(Table 4.10).

3. **FT_CBPA.ipynb**: Implementation of the integrated baseline combining payload-aware allocation with the R3P protocol (Table 4.25 and Table 4.26).
4. **FT_ACO_BA.ipynb**: Realization of the heuristic-based integrated baseline involving ant colony optimization and bat algorithms(Table 4.17 and Table 4.18).
5. **SPA_CBPA.ipynb**: Full implementation of the proposed proactive successor-based recovery protocol.
6. **SPA_CBPA_Ablation_Study.ipynb**: Systematic evaluation of successor selection policies (Static vs. Dynamic) as discussed in the ablation analysis(Table 5.4).