Choros: A Distributed Coordination Framework for Multi-Robot Botball Systems

Ankush Ahuja*, Leopold Kernegger
WeThePeople
Technologisches Gewerbemuseum
Vienna, Austria
*Corresponding author's email: ankush@ahuja.app

***Abstract*—Choros is a lightweight and efficient coordination framework for autonomous robots competing in the Botball educational robotics competition. Instead of programming low-level behavior sequences, Choros enables a declarative model of robot control, where users define desired outcomes rather than individual actions. Game tables are serialized into graph representations, and robot positions are tracked and shared in real time, enabling precise goal specification, dynamic task planning, and intelligent collision avoidance. Designed for simplicity and robustness, Choros empowers teams to focus on strategy and reliability while maintaining low system overhead.**

***Index Terms*—Robotics, Multi-agent systems, Distributed coordination, Path planning, Botball**

I. Introduction

Botball is an international educational robotics competition in which autonomous robots are required to complete a sequence of tasks on a predefined game table. These tasks often demand precise navigation, coordination, and interaction with objects in a dynamic and spatially constrained environment. A critical component of success in Botball is accurate positioning—poor navigation can result in missed objectives, hardware damage, or complete failure of a competition run.

Numerous research efforts have already explored methods for enhancing the precision of robot movement on the Botball field. This paper builds upon findings such as those presented in [1], which focuses on improving motion accuracy through calibration and control techniques. However, rather than attempting to further refine motion accuracy itself, *Choros* takes a different approach: it focuses on building a high-level, declarative navigation and coordination framework that assumes reliable low-level motion primitives.

A. The Game Table Environment

The Botball game table, provided by KIPR, features a standardized white surface with black electrical tape arranged in both horizontal (0°) and vertical (90°) alignments. These black lines serve as paths for line-following and are always continuous, allowing for intersections—an essential property for graph-based modeling. Robots navigate this field using reflective line sensors known as *Tophat Sensors*, which are included in the official Botball kit.

These sensors operate by emitting light and measuring its reflection from the surface below. White surfaces reflect light strongly, while black lines absorb most of the emitted light. This binary distinction enables the robot to distinguish between line and non-line surfaces, facilitating accurate line-following behavior.

B. Line Following

Line following is a foundational method for reliable movement in Botball. By continuously adjusting the robot's movement based on sensor readings, the robot can drive along the black tape with precision. Because lines on the field are always aligned to the grid and continuous, they naturally form intersections—locations where horizontal and vertical paths cross or meet. This structural property of the game table will be of great importance later in the design of the navigation system.

C. Square Ups

Another critical application of Tophat sensors is *squaring up*: a technique where both left and right sensors are aligned on a black line to correct the robot's heading and lateral drift. While line-based squaring offers reasonable accuracy, a more precise variant known as *wall square up* uses two physical bumper switches on the front of the robot. When both buttons are pressed simultaneously against a wall, the robot can reset its pose alignment with high precision.

D. Sensor Feedback and PID Driving

In addition to physical feedback mechanisms, KIPR's controllers include a built-in gyroscope. As discussed in [1], this gyroscope allows for accurate heading measurements, enabling robots to perform controlled turns and maintain straight paths using a PID (Proportional–Integral–Derivative) controller. This greatly enhances the consistency of autonomous driving by correcting for motor imbalances and slippage in real time.

E. From Imperative to Declarative Control

By leveraging the structure and properties of the Botball game table—such as line alignment, intersections, known distances, and sensor feedback—one can design a declarative coordination system for autonomous robots.

Rather than tightly planning each route and scheduling tasks manually to avoid collisions, the system allows users to define *what* should happen—i.e., which tasks should be completed and where—while leaving the exact *when* and *how* to the coordination logic.

Tasks, each bound to a location and optionally dependent on the completion of other tasks, can be treated as part of a larger lifecycle. This enables the system to automatically adjust exe-

cution order, resolve dependencies, and prevent conflicts with other robots in real time. The result is a flexible and robust framework that supports dynamic, multi-robot coordination on a shared game field.

II. Terminology and Algorithms

This section introduces fundamental data structures and algorithms used throughout the system described in this paper. While their specific applications will be discussed later, the following definitions and procedures are essential to understanding the framework's internal logic and decision-making processes.

A. Graphs

A *graph* is a data structure consisting of a set of *nodes* (also called *vertices*) and *edges* connecting pairs of nodes. Graphs are widely used to model relationships, connectivity, and structure in both physical and abstract systems.

This paper specifically uses *directed graphs*. In a directed graph, each edge has a direction—it connects a source node to a destination node and not the other way around. This directionality allows graphs to represent one-way relationships, such as movement constraints, causality, or precedence.

Each edge in a graph may also be assigned a *weight*, representing a cost, distance, time, or any other metric associated with traversing that edge. Weighted, directed graphs are the foundation for many types of planning and scheduling problems.

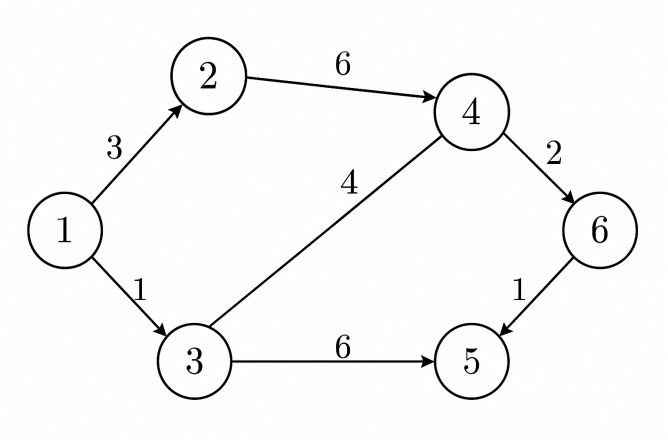

Fig. 1: A directed, weighted graph.

B. Pathfinding

Pathfinding refers to the process of determining the optimal path between nodes in a graph. The goal is typically to minimize the total cost accumulated along a path, where costs are derived from edge weights.

One of the most well-known pathfinding algorithms is *Dijkstra's algorithm* [2]. Dijkstra's algorithm computes the shortest paths from a single source node to all other nodes in a graph with non-negative edge weights. Dijkstra's algorithm guarantees optimal paths.

C. Dependency Resolution

Dependency resolution addresses the problem of ordering tasks when certain tasks depend on the completion of others. This problem can be modeled using a *directed acyclic graph* (DAG), where each node represents a task and each directed edge indicates that one task must be completed before another can begin.

Because DAGs contain no cycles, they have at least one valid *topological ordering*: a linear sequence of tasks that satisfies all dependency constraints.

A common method to resolve dependencies in a DAG is a *breadth-first search* (BFS)-based topological sort [3]. The algorithm proceeds as follows:

1) Identify all nodes with zero incoming edges—these represent tasks that can be executed immediately.
2) Place these nodes into a queue.
3) Repeatedly dequeue a node, append it to the sorted result, and remove its outgoing edges from the graph.
4) If any neighboring nodes now have zero incoming edges, enqueue them.
5) Continue until all nodes are processed or a cycle is detected (in which case no valid ordering exists).

This approach ensures that each task is executed only after all its dependencies have been resolved. The resulting order can then be used to safely execute or schedule tasks in complex workflows.

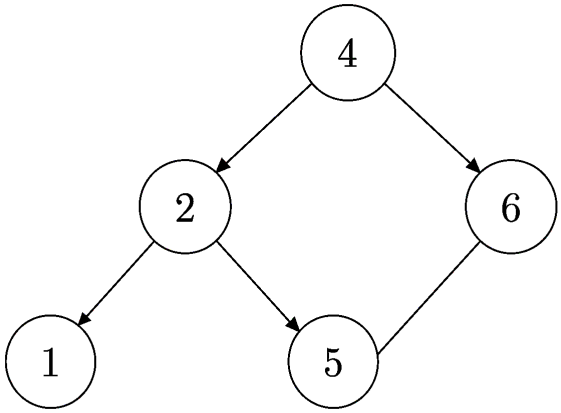

Fig. 2: A directed acyclic graph (DAG) illustrating task dependencies. Each node represents a task, and directed edges represent prerequisite relationships.

III. The Navigation System

This section outlines how navigation works in its simplest form—without the involvement of a second robot. All mechanisms described here form the basis of *Choros* and assume a static environment where coordination is not required.

A. Graph-Based Representation

To enable declarative navigation, the physical layout of the Botball game table is abstracted as a *directed, weighted graph*. In this model, each *primary node* corresponds to a line intersection between two black tape segments. The straight black lines between intersections form the *edges* of the graph.

All edges are *directed*, meaning they encode a specific direction of traversal. This is not due to mechanical limitations but because each edge corresponds to a unique orientation in the global coordinate system. For example, movement from node A to B may occur at 0°, while movement from B to A takes place at 180°. Although the physical distance between nodes is symmetric, the direction of travel is not, making it essential to treat each edge as a separate, directed connection.

B. Primary Node Navigation

A *primary node* is a location where two black lines intersect orthogonally on the white game table surface. Navigation between these nodes is achieved through *line following*. The robot is equipped with two *Tophat sensors*: one is used to track the current line, while the second is positioned to detect new lines orthogonal to the direction of travel.

While following a line, the outer sensor constantly checks for the presence of a perpendicular black line. When detected, it signals that an intersection has been reached—thus identifying the arrival at a new primary node. The robot then updates its internal graph position and prepares for the next leg of the route.

Each transition from one node to another is associated with a cardinal direction (e.g., North, East, South, West), mapped to a corresponding rotation angle. This allows the robot to align itself correctly when transitioning between non-parallel segments.

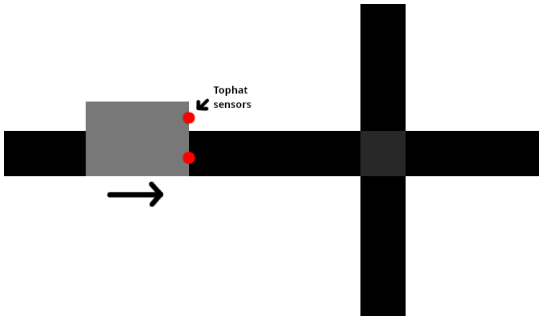

Fig. 3: Primary node navigation.

C. Secondary Node Navigation

A *secondary node* occurs when a black line terminates at the edge of the game table without forming an intersection. These endpoints are still useful for navigation and can be reached reliably through a *wall square-up*.

The table edges are enclosed by PVC pipes. When a robot follows a black line to such a boundary, it continues driving until both of its bumper switches are triggered. This contact ensures the robot is aligned straight against the wall. Once this is achieved, the robot registers its position as a secondary node and can proceed to rotate or backtrack as needed.

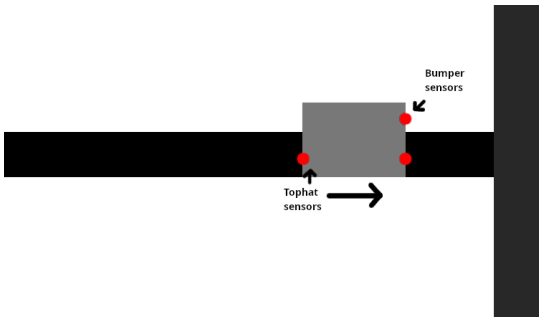

Fig. 4: Secondary node navigation (the right side illustrates a wall/pvc-pipe).

D. Declarative Path Planning

Rather than hardcoding each movement, *Choros* accepts a goal in terms of a target node. The system computes the shortest sequence of nodes needed to reach that goal using *Dijkstra's algorithm*.

Dijkstra's algorithm treats the graph as a set of weighted paths and iteratively explores all reachable nodes, always expanding the route with the lowest total cost. The output is a path composed of node transitions, each corresponding to a straight-line movement between two waypoints.

To execute this path, the robot:

- Computes the relative turn angle needed to align with the next edge.
- Follows the line segment to the next node, using square-up techniques as necessary.
- Repeats this process until the destination node is reached.

This declarative system simplifies both programming and error handling, as it abstracts away the low-level motion logic into a repeatable and well-tested procedure.

E. Locations

While nodes serve as reliable waypoints, most game elements —such as scoring cups or dispensers—are not placed directly on nodes. To bridge this gap, *Choros* introduces the concept of a *location*: a precise target defined relative to a node but positioned off the navigation graph.

Each location is described as a list of one or more *vector offsets*. A vector offset consists of an x and y displacement (in millimeters) from the last position. Internally, each vector is converted into a direction (angle) and scalar (distance), which are used to guide the robot's movement.

To reach a location, the robot performs the following steps:

1) It rotates to the correct direction using the onboard gyroscope.
2) It then advances by a calibrated distance using the *BackEMF* tick counter.

This mechanism allows *Choros* to reach targets not aligned with primary or secondary nodes, such as free-floating game objects or zones. In cases where a direct path is obstructed, multiple vectors may be chained to navigate around obstacles while still following the shortest feasible route.

As discussed in [1], movement driven purely by BackEMF introduces cumulative drift, especially across longer segments. For this reason, vectors should be kept short, and locations

should be placed as close to nodes as possible. Unlike nodes, which can be reached and confirmed using square-ups, locations depend entirely on accurate angle rotation and tick-count calibration, making them the most error-prone and sensitive stage in the navigation process.

IV. Networking and Coordination

This section describes the communication layer used in *Choros* to enable safe multi-robot operation. The system relies on frequent, lightweight broadcasts to share intent and position between autonomous agents, allowing for real-time collision avoidance without the need for centralized scheduling.

A. Heartbeat Transmission

Each robot periodically transmits a *heartbeat message* using the User Datagram Protocol (UDP). These messages are sent approximately every 200 milliseconds and contain a JSON-encoded snapshot of the robot's current intent.

The payload includes:

- The identifier of the *current node* the robot occupies.
- A list of *planned future nodes* that make up the robot's intended path.

The use of UDP is deliberate. Since each heartbeat is stateless and short-lived, occasional packet loss is tolerable. New updates quickly replace outdated ones, making the protocol efficient and resilient for real-time environments.

B. State Sharing and Awareness

Upon receiving a heartbeat, each robot updates its internal model of the other robot's state. This includes:

- The other robot's current node.
- The complete list of nodes it plans to traverse.

This shared information forms the basis for distributed decision-making: each robot remains aware of the other's intent and can proactively avoid path conflicts.

C. Path Claiming and Blacklisting

When a robot initiates path planning (e.g., via Dijkstra's algorithm), it *blacklists* any node currently included in the other robot's future path. These nodes are considered temporarily *claimed* and cannot be used unless released.

A critical safety rule applies: a node is not considered free the moment the other robot departs it—it becomes available only when the other robot has *reached the next node* in its path. This prevents overlap during acceleration, turning, or communication delays.

D. Decentralized Conflict Avoidance

This system follows a simple and effective policy: *first-come, first-served*. The first robot to claim a node owns it until it progresses. If the other robot encounters a conflict while planning, it must:

- Reroute to avoid the claimed nodes, or
- Wait and retry after the next heartbeat update.

This approach eliminates the need for centralized arbitration and ensures collision avoidance purely through shared knowledge and local planning.

E. Error Handling and Fallback Strategies

Despite the system's lightweight design, communication is never guaranteed. UDP packets can be dropped, robots may reboot, or environmental interference may block messages. In such cases, the assumption that robots can rely solely on live coordination no longer holds.

For this reason, *Choros* does not eliminate the need for planning altogether. Developers must be aware that coordination may fail and should prepare appropriate fallback strategies.

Recommended approaches include:

- Sectoring the game table: dividing the field into distinct regions assigned to each robot to prevent overlap. This spatial isolation minimizes the chance of collisions even when communication is lost.
- Alternative lifecycles: preparing simplified, non-coordinated behavior sequences that can be activated if the robot detects communication loss or failure to claim a path.
- Timeout-based logic: introducing thresholds after which missing heartbeats are treated as a fault condition, prompting a transition into recovery behavior or safe halting.

These strategies are not part of the *Choros* implementation due to the vastly different requirements and constraints of fallback designs.

V. Lifecycle and Task Management

The lifecycle of a robot in the *Choros* system encapsulates every phase of its operation—from initial setup to shutdown. While a lifecycle may appear linear in nature, its internal structure supports robust, dynamic behavior capable of adapting to failures and execution variability.

A. Lifecycle Phases

The typical lifecycle of a robot run in *Choros* consists of the following steps:

1) `calibrate()` — Initializes the robot's sensors and actuators. This may include gyroscope zeroing, Back-EMF calibration, and alignment with the game table.
2) `wait()` — Pauses execution until the starting light is detected or a start condition is met.
3) `execute_tasks()` — Begins execution of high-level tasks managed through the internal dependency graph.
4) `clean()` — Executes any cleanup steps such as stowing arms or retracting mechanisms.
5) `reset()` — Resets internal state in preparation for testing, reruns, or debugging sessions.

Each of these stages is executed under the authority of the *lifecycle controller* which maintains references to key subsystems such as the navigation logic, sensor interfaces, motor controllers, and task manager.

B. Tasks as a Dependency Graph

Tasks in *Choros* are not executed in a fixed sequence. Instead, they are modeled as nodes in a *directed acyclic graph* (DAG),

where each edge defines a prerequisite relationship between tasks. A task may only be executed once all its parent (dependency) tasks have either completed successfully or are marked as not required.

This structure allows for flexible execution strategies. For example, if a task unexpectedly fails, it may be rescheduled for later execution, while downstream tasks that do not depend on it may proceed unhindered. This results in a *self-regulating execution flow* that dynamically adapts to runtime outcomes.

Each task returns an integer status code:

- `> 0` — Task completed successfully.
- `= 0` — Task failed but may be reattempted.
- `< 0` — Task failed definitively and will not be retried.

C. BFS-Based Resolution and Queue Management

The lifecycle uses a *breadth-first search* (BFS) strategy to identify which tasks are eligible for execution at any given point. These tasks are placed into a *task queue*, which is processed serially:

- Tasks at the front of the queue are executed first.
- On success, any adjacent tasks with all dependencies satisfied are added to the end of the queue.
- Tasks that fail with reattempt allowed are also requeued.

This mechanism ensures that tasks are performed in a dependency-safe order, and that failures do not stall the entire process unless they block other critical operations.

D. Declarative Benefits

Because tasks are defined declaratively and reference goals rather than step-by-step instructions, *Choros* can adapt fluidly to changing execution contexts. The navigation system, for example, does not require predefined movement scripts. If a task requires reaching a location, the system simply computes a path from the robot's current position—regardless of how it arrived there.

This is in contrast to traditional imperative strategies where task order and routing are tightly coupled. In such systems, a task failure often invalidates all downstream assumptions. In *Choros*, however, the system adapts: tasks are retried, reordered, or skipped as needed, and navigation is recomputed on demand.

This approach significantly reduces the complexity of preplanning while increasing robustness. Developers describe **what** must be done; the lifecycle decides **when** and **how** to do it.

E. Lifecycle Encapsulation

The lifecycle itself is a container object that holds everything the robot needs during runtime:

- Shared state (e.g., flags)
- Subsystem references (navigation, motors, sensors)
- The task graph and current task queue

Each task receives a pointer or reference to the active lifecycle, allowing it to query or manipulate global state as needed.

VI. The Big Why

In the chaos of real-time robotics competitions like Botball, students and developers often face the challenge of building systems that are not only functional but also adaptable, maintainable, and fault-tolerant. *Choros* addresses this challenge by introducing a unifying structure for perception, navigation, task planning, and execution.

At its core, *Choros* is not a single algorithm or feature—it is a methodology. It encourages teams to stop hardcoding fragile, linear control flows and instead embrace declarative principles that reflect how complex systems should behave in uncertain environments.

A. Unified Architecture, Modular Components

By abstracting the game table into a navigable graph, separating navigation from task intent, and introducing the concept of node-relative *locations*, we enable reliable movement across the field. This structure is further enhanced by the lifecycle system, which orchestrates the entire run—from sensor calibration and waiting for light to executing task DAGs.

The resulting architecture is both modular and composable:

- Navigation logic is graph-based and declarative.
- Tasks are self-contained and schedule-aware.
- Lifecycle encapsulates global state, hardware references, and coordination interfaces.

B. Rethinking Task Complexity

This architecture shifts a significant portion of complexity into the design of each task. Since the system handles navigation and scheduling declaratively, the burden moves toward making each task *robust and self-aware*. Developers must clearly define:

- what conditions must be satisfied before a task can execute,
- what constitutes a success or recoverable failure,
- and how the task should react in edge cases or when a dependency fails.

Tasks become less about scripting behavior and more about *declaring constraints and assumptions*—a change that requires deeper forethought, but results in a far more maintainable and adaptable system.

C. The Result

The end product is a resilient and maintainable robot behavior model that not only performs reliably but is also easier to debug, extend, and reuse. The system is reactive to unexpected failures, agnostic to execution order, and centered around clear data-driven decisions rather than procedural scripts.

Choros shows that high-level robotics frameworks can be both simple and powerful—provided that the underlying structure is well thought-out and grounded in abstraction, not improvisation.

References

[1] B. Klauninger, K. Lindorfer, S. Kawicher, T. Koch, V. Griesmayer, and L. Kornthaler, "Enhancement of Accuracy in Botball Navigation," 2022, [Online]. Available: https://robo4you.at/static/3964d6b8b8c801fcec42c695f641b120/enhancement-of-accuracy-in-botball-navigation.pdf

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959, doi: 10.1007/BF01386390.

[3] "Topological Sorting Using BFS." 2025.