Cooperative Collision Avoidance in Multi-Agent Systems for Autonomous Educational Robotics

Maximilian Paesold
Department of IT
Technologisches Gewerbemuseum
Vienna 1200, Austria
mpaesold@student.tgm.ac.at

***Abstract*—Autonomous robot competitions require a high level of operational reliability. However, when multiple agents have overlapping working spaces, independent navigation leads to a high likelihood of collisions. This paper presents a cooperative collision avoidance system utilizing a TCP/IP-based synchronization protocol. The architecture, written in C for the KIPR Wombat, uses non-blocking sockets to allow for real-time communication for decision making, while also not freezing an agent's primary control loop.**

***Index Terms*—Multi Robot Systems, Collision Avoidance, Asynchronous Communication, Educational Robotics, Botball**

I. Introduction

At the 14th European Conference for Educational Robotics (ECER2026 [1]), students are challenged to apply a wide range of engineering knowledge. In the Botball competition format, teams use two robots, powered by a Wombat controller [2], to solve different logistical tasks in varying complexity. While the most crucial part of a match is the individual performance of a robot, the interaction between these agents has a comparably relevant impact on the outcome of a run.

The usual approach at the ECER is to treat each robot as an isolated entity. However, for more complex tasks, the need to use the same physical space increases. In such cases, ignoring communication as a sensory data source increases physical collisions drastically, becoming a major bottleneck for competition performance. This paper demonstrates our team's solution to this vulnerability.

II. State of the Art

A. Related Work on Multi-Agent Coordination

Prior work on multi-agent coordination identifies communication as a fundamental enabler of reliable cooperative behaviour [3], [4]. Cao et al. show that even minimal, lightweight communication between robots outperforms purely sensor-based independent navigation in scenarios where multiple agents share the same physical space [3]. Parker demonstrates that reliable multi-robot systems must continuously monitor their partners' status and adjust their behaviour upon partner failure [4]. These principles are directly reflected in our communication design. However, in the specific context of educational robotics, such approaches have not been formally applied to competitive environments like Botball, where limitations on hardware and Wi-Fi dependency introduce challenges not addressed by existing approaches.

B. Observed Performance Gaps in Botball Competitions

To identify the most common sources of scoring inconsistency, informal interviews were conducted with three teams from previous ECER editions: one team from ECER2023 and two teams from ECER2025. Two out of three teams, unprompted, mentioned inter-agent collisions as a direct cause of failed runs. The largest team reported that 2 out of 3 seeding runs failed due to robot-robot collisions, and that collisions were a contributing factor in a significant portion of the bottom half of seeding scores observed across competitors, though these also include collisions with game board elements. Low battery levels and motor drift were cited as the other major sources of run failures. The team from ECER2023 acknowledged collisions as an issue when specifically asked. These findings indicate that inter-robot collision was a recurring cause of failures across multiple teams and competition years, alongside hardware-related issues such as motor drift and low battery levels.

The most common approach to solving problems on the game table (and therefore scoring) was to view each bot as an independent player with no communication between one another. Due to most tasks being solvable by just one bot, this approach seems sufficient at first glance. This line of reasoning, though, leaves out many factors that play into a run of a Botball game, most importantly consistency in hardware and environment. This is further supported by the past tournament scores [5] which reveal a significant performance gap (Fig. 1). While teams reached a peak average of 200.3 points, the average Seeding Total was only 136.7, despite each team's lowest scoring run being omitted in the calculation. If these numbers are taken into account, the true tournament average drops to 98.7 points. While some of these losses can be attributed to mechanical failures, the interview findings suggest that collisions between agents were a notable contributing factor.

To address these reliability issues, this paper proposes the use of a bot synchronization protocol based on a client-server architecture. Instead of a bot choosing its actions independently based only on its own sensors, it also considers data sent from the second bot.

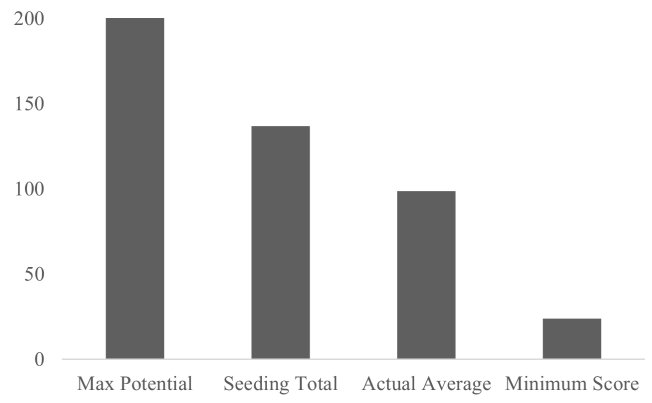

Fig. 1. Tournament performance metrics showcasing the discrepancy between peak agent potential and actual match averages due to reliability issues.

III. Concept / Design

In this section, we present the communication architecture design of our synchronization protocol. Our focus in the design process was to keep the modularity of being able to work on just one bot at a time but also gain the reliability from having access to communication between bots.

A. Client-Server Model

Since relying on an external server to pass messages is prohibited by the game rules [6], we instead use a direct peer-to-peer TCP/IP connection. In our case, this means hardcoding one agent to act as the Server, listening on a specific port, while the other acts as the Client connecting to the host's static IP.

B. Asynchronous Polling and Non-blocking I/O

A critical design element in our communication architecture is the switch from blocking to non-blocking sockets. By default, TCP sockets are in "blocking" mode. This means that the bot will pause its action until it receives an answer. For example, if we try to use `recv()` while having the sockets set to blocking, but no message comes through, the agent would freeze, and we would lose control of the bot. If we instead set the ports to non-blocking, the program would just return to the main-loop. Therefore we differentiate between two phases:

Initialization Phase:
During pre-match setup, the socket commands `bind()`, `listen()`, `accept()`, and `connect()` are all set to blocking. This ensures both bots are fully powered and connected to the network by enforcing a mandatory handshake before the starting light even fires. The program execution will not proceed unless a successful TCP handshake is completed.

Execution Phase:
As soon as the connection has been established, the system goes into non-blocking mode. In robotics, this is necessary, so that we retain motor and sensor control even if the partner hasn't sent data right away. This way a bot can simultaneously poll the network buffer for status updates and also perform high attention tasks like line following.

C. Token-Based Synchronization

The system does not continuously send a heavy stream of unnecessary positional data (like exact x/y coordinates). Instead, it uses a much lighter event-token-based protocol. Whenever an agent reaches a designated area, for example characterized by one of the black tape lines on the game table, it transmits a simple string token, the simplest version just being a `"Here"`. The partner agent who is polling its receiving buffer will then act on this token accordingly. The following example illustrates the protocol in practice:

Collision avoidance at game start: The limitations set by the Botball hardware introduce variability even at the simplest tasks such as driving straight ahead. Our game strategy requires both of our robots to start in the same starting box while also arriving at their destination quickly. Instead of risking collision, we let one bot wait in the starting box, polling for incoming data. Meanwhile, the other bot starts driving. When it reaches a destination hardcoded by us (e.g. the black tape on the gameboard), it sends a token to its waiting partner: `"AtBlackTape"`. The partner may then start safely driving without the possibility of collision. In this example, the waiting bot is stationary, but this does not have to be the case, as made possible by the non-blocking ports.

Currently, the protocol is designed for exactly two agents. Extending to more agents would introduce additional networking complexity, such as the need for broadcast mechanisms or a dedicated coordinator, and is left for future work.

D. System Level Failsafes

In order to avoid major problems when facing network degradation, we built a failsafe for such cases. If the TCP connection is severed mid-match (e.g. a WI-FI dropout causing `recv()` to return 0), the system is designed to catch the exception and close the socket. Depending on the game state (evaluated by looking at the last received message), the agent may then decide to proceed with its current objective or transfer to a safe strategy to avoid crossing paths with the other bot and score safer points instead.

The system assumes a sufficiently low Wi-Fi latency throughout the match. During periods of partial network degradation, the non-blocking architecture ensures agents continue executing while polling for delayed tokens, affecting match timing but not causing robots to behave incorrectly. Full disconnection is handled as above.

E. Event Stack

For our current strategy for ECER2026, we are using an expanded implementation of the simple event-token transfer. Each bot stores a hardcoded event stack corresponding to the expected order of messages it will receive from its partner. To differentiate between expected and unexpected states, the sending bot compares its current sensor state against its own hardcoded event sequence and prefixes the token accordingly:

e_ indicates normal progress, u_ signals an unexpected game state.

The event stack serves three purposes: confirming that a shared zone has been cleared and is safe to enter, localizing where in the game sequence a failure occurred to enable position-aware fallback decisions, and detecting if an expected token never arrives. Since tokens are transmitted in the fixed order defined by the sending bot's code sequence, it means that expected tokens will always arrive in the correct order.

Cooperative ramp strategy: The 2026 ECER game table features an upper deck area accessible via a ramp [6], where removing poms yields a significant point bonus, further amplified by placing both bots into that area. Our strategy sends the first bot up the ramp as part of its normal path. Once it reaches the upper area and clears the ramp, it sends an `e_RampClear` token, signalling the second bot to follow. If the first bot fails earlier in its course, it transmits a `u_`-prefixed token. If it does so from a position where it cannot block the ramp, the second bot can drive up independently instead. This way, the ramp stays accessible in most failure cases, and at least one bot reaches the top.

IV. Implementation

This section will cover the practical implementation of our synchronization protocol, which was written in the C programming language for the KIPR Wombat platform.

A. Network Communication Driver

The following C code implements the connection establishment and the transition from a blocking initialization handshake to a non-blocking state.

Listing 1. TCP/IP Peer-to-Peer Synchronization Driver

```
// Configuration for Peer-to-Peer TCP
#define IS_SERVER 1 // Set to 0 for Client bot
#define SERVER_IP "10.200.0.70"
#define PORT 500

static int comm_fd = -1;

// Transition socket to non-blocking mode for real-
    time control
void set_nonblocking(int fd) {
   int fl = fcntl(fd, F_GETFL, 0);
   fcntl(fd, F_SETFL, fl | O_NONBLOCK);
}

int comm_init() {
   struct sockaddr_in addr = { .sin_family = AF_INET
       , .sin_port = htons(PORT) };

   #if IS_SERVER
      int srv = socket(AF_INET, SOCK_STREAM, 0);
      int opt = 1; setsockopt(srv, SOL_SOCKET,
          SO_REUSEADDR, &opt, sizeof(opt));
      addr.sin_addr.s_addr = htonl(INADDR_ANY);
      bind(srv, (struct sockaddr*)&addr, sizeof(addr
          ));
      listen(srv, 1);
      // Blocking Handshake: Enforces
          synchronization before start
      comm_fd = accept(srv, NULL, NULL);
      close(srv);
   #else
      comm_fd = socket(AF_INET, SOCK_STREAM, 0);
      inet_pton(AF_INET, SERVER_IP, &addr.sin_addr);
      // Blocking connection: Ensures partner is
          ready
      if (connect(comm_fd, (struct sockaddr*)&addr,
          sizeof(addr)) < 0) return 0;
   #endif

   set_nonblocking(comm_fd); // Switch to
       asynchronous execution
   return 1;
}

void signal_here() {
   if (comm_fd >= 0) send(comm_fd, "HERE\n", 5, 0);
}
```

To maintain code modularity requirements, we wrote several high-level functions. Simpler versions of those can be seen in the above code. Our approach was to center the driver on a dual-phase socket state transition. During the initialization phase, blocking calls, specifically `accept()` for the bot acting as the server and `connect()` for the bot acting as the client, are used to establish the TCP/IP link. After a successful handshake, the socket is reconfigured to non-blocking using `fcntl()` utility with the `O_NONBLOCK` flag.

For later implementation one can simply use `send()` (e.g., `send(comm_fd, msg, (int)strlen(msg), 0);`) or `recv()` (e.g. `ssize_t n = recv(comm_fd, buf, sizeof(buf)-1, MSG_DONTWAIT);`) to transfer data between bots.

V. Evaluation

A. Figure-8 Synchronization

In order to test our program we decided to create a scenario which mandated the utilization of a means of communication to find a solution. Two bots have to move along the black tape according to the arrow's direction shown in Fig. 2. They alternate between going left and right after the middle section. We used a simple line following algorithm for pathing, combined with our synchronization protocol, in order to allow two bots to go through the center in the same loop. By making a bot wait at the turn before the shared intersection and sending a `"Here"` string while also polling for that same incoming message, it would only continue if the partner bot was also at its last turn before the shared intersection (on the opposite side). This way, we could use two basically identical bots for the track, with the only difference being that we assigned the bots a different `PRIORITY_ROLE`: `PRIORITY_ROLE` = 1 means that a bot would continue driving as soon as synchronization was complete, while `PRIORITY_ROLE` = 0 means that it would wait for 5 seconds before also continuing.

B. Results

Testing was conducted over two sessions, containing around 20 timed runs of 2-3 minutes each after consistent line following was established. No communication failures were observed across any run. Approximately 10% of runs failed due to line following errors, specifically, inconsistent turn radii, causing a

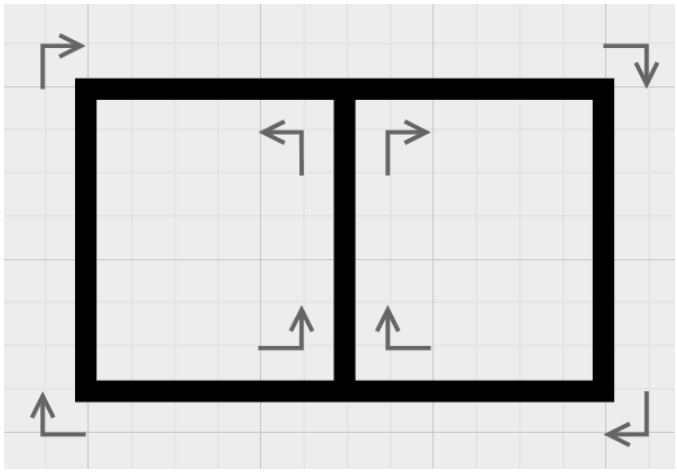

Fig. 2. The Figure-8 test scenario. The central shared path represents a zone where the multi-agent synchronization protocol is essential to preventing collisions.

bot to lose the line. When this happened, the partner bot waited at its checkpoint for the duration of the run, as unexpected-state tokens were not implemented in this test. This limitation is addressed in the full game strategy through the use of `u_`-prefixed tokens and position-aware fallback decisions, as described in Section III-E. No collisions occurred in any run where the synchronization protocol was active.

A baseline comparison was conducted using a fixed time/distance offset instead of synchronization to evaluate the benefits of the protocol. In this configuration, around 80% of runs failed due to collision within 2 figure-8 loops. These failures were either caused by a direct collision in the shared segment or differences in speed causing the bots to converge. Our best-performing run lasted for over 20 minutes.

VI. Conclusion

In this paper, we have proposed a solution to the lack of consistency in using two autonomous robots on the ECER2026 game table. By moving away from independently navigating agents to a communication framework that uses simple event-tokens, we have directly addressed agent collision, a major issue in past events. Our Figure-8 test results demonstrate zero collisions across all synchronized runs, with stable operation maintained for over 20 minutes, compared to an 80% collision rate within 2 loops using a time-offset baseline. These results prove that this software architecture provides a reliable foundation for more complex, real-time robotic tasks.

VII. Outlook

The current implementation is limited to two agents communicating over a single peer-to-peer TCP connection. Extending the protocol to support more than two agents would require a more general communication topology, such as a broadcast mechanism or a dedicated coordinator node, along with a revised event stack design to handle multiple concurrent token streams.

A further goal is the integration of the synchronization driver into RaccoonLib [7], an open-source robotics platform for Botball developed at HTL St. Pölten. While RaccoonLib currently provides a transport layer for inter-process messaging, it does not yet include a module for inter-robot communication over Wi-Fi. Integrating the synchronization protocol as a dedicated module would extend the platform's capabilities and make cooperative collision avoidance accessible to other ECER teams without requiring them to implement low-level socket management themselves.

References

[1] PRIA, "ECER 2026: 14th European Conference on Educational Robotics," *Practical Robotics Institute Austria*, 2026. [Online]. Available: https://ecer.pria.at/ [Accessed: Apr. 09, 2026].

[2] KIPR, "Botball Hardware and Software Overview," *KIPR Resources*, 2026. [Online]. Available: https://www.kipr.org/kipr/hardware-software [Accessed: Mar. 14, 2026].

[3] Y. U. Cao, A. S. Fukunaga, and A. B. Kahng, "Cooperative Mobile Robotics: Antecedents and Directions," *Autonomous Robots*, vol. 4, no. 1, pp. 7–27, 1997.

[4] L. E. Parker, "ALLIANCE: An Architecture for Fault Tolerant Multirobot Cooperation," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 2, pp. 220–240, 1998.

[5] KIPR, "2025 Botball Tournament Scores," *KIPR Resources*, 2025. [Online]. Available: https://www.kipr.org/2025-botball-tournament-scores [Accessed: Apr. 09, 2026]. Tournament data: https://docs.google.com/spreadsheets/d/1-_6ecoKuBGMElU08Z7WBsIf00JXdasVZfwQC8TqYlk4/edit?gid=1743140526#gid=1743140526 [Accessed: Apr. 09, 2026].

[6] PRIA, "Botball Game Documents and Rule Amendments for ECER 2026," *Practical Robotics Institute Austria*, 2025. [Online]. Available: https://ecer.pria.at/documents/2026/botball/game-documents [Accessed: Apr. 06, 2026].

[7] T. Madlberger et al., "RaccoonLib: Core Robotics Library for RaccoonOS," 2026. [Online]. Available: https://github.com/htl-stp-ecer/raccoon-lib [Accessed: Apr. 09, 2026]