ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

Master's Degree in Computer Science and Engineering

Lowering the Reality Gap in Aggregate Programs Validation: Running Collektive Over Unity

Thesis in:
Software Process Engineering

Supervisor
Prof. Danilo Pianini

Candidate
Filippo Gurioli

Co-supervisors
Martina Baiardi
Angela Cortecchia

Graduation Session: IV
Academic Year 2024-2025

Abstract

Max 2000 characters, strict.

To my grandparents and Roberto...

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Modern computing is moving away from the era of powerful and isolated machines toward one composed by massively interconnected ensembles of devices. We can observe this transition everywhere, from global cro:IoTInternet of Thing (IoT) sensor networks to smart city infrastructures. In such scenarios, the focus shifts from 'how to compute' to 'how to coordinate'.

As the number of devices in these systems grows into the thousands or millions, traditional centralized management becomes a bottleneck. The latency, bandwidth constraints, and single-point-of-failure risks of a 'command-and-control' architecture make it unsuitable for the dynamic, often unpredictable environments these systems inhabit. Instead, we must look toward decentralized coordination, where collective intelligence arises from local interactions rather than global oversight.

This thesis explores the intersection of high-level collective programming and high-fidelity simulation. Specifically, it addresses the engineering gap between abstract coordination models, such as cro:ACAggregate Computing (AC), and the practical requirements of developing, testing, and deploying these models within realistic 3D environments. By leveraging the power of modern game engines and automated development workflows, this work aims to provide a robust infrastructure for the next generation of collective system design.

1.1 Motivation: Swarm Behaviour

The natural world provides the strongest precedence for the goal of resilient decentralized coordination. From the coordinated flashing of fireflies to the intricate architectural achievements of termite mounds and the smooth collective motion of starling murmurings, biological systems exhibit an efficiency that is frequently difficult for classical engineering to match. These phenomena, which are collectively referred to as cro:SISwarm Intelligence (SI), arise from the interaction of many simple agents that follow localized rules rather than from a global supervisor.

In a natural swarm, intelligence is inherently distributed and emergent. Individual agents (be they ants, bees or birds) possess only a partial perception of their surroundings. The collective however can solve high-order problems such as finding the shortest path to a food source or executing rapid evasive maneuvers against predators. From an engineering perspective, these systems offer three indispensable properties:

- the absence of a central controller; the loss of individual units does not compromise the mission.
- The logic governing ten agents often remains functional for ten thousand, as interactions remain local regardless of total population size.
- Swarms autonomously adapt to dynamic environments, re-configuring their behaviour in response to external stimuli.

As we attempt to port these characteristics into the digital and physical domains (specifically through paradigms like AC) we face a significant translation gap. While the mathematical models for collective logic are maturing, the infrastructure to test them in realistic, high-fidelity environments remains fragmented. To truly harness the potential of swarm behaviour in human-made systems, we must develop tools that can simulate the complex interplay between decentralized algorithms and the physical world.

1.2 Problem Statement: Engineering Challanges in Simulation

Simulation has been widely explored in terms of scalability, but not many researches have been done regarding high-fidelity. This field brings into play hard constraints that mathematical rigor often does not consider. Physics collisions, gravity and friction are just examples of what a good high-fidelity simulator could add to a cooperative swarm simulation. Traditional simulators often prioritize the number of agents at the expense of environmental complexity, leading to a 'reality gap' that complicates the deployment of algorithms onto physical hardware. Fortunately, game engines do this work for us; they add physics engines capable of computing the result of physical interactions with rigor. The real problem now becomes only one: bridging these two worlds.

The challenge of bridging high-level coordination with game-engine-driven physics is not merely a matter of data transfer, but one of architectural alignment. In particular:

- synchronism: collective programming models rely on discrete logical steps whereas game engines operate on a continuous, high-frequency tick (e.g. 60Hz, 60 frames per second).
- Abstraction: collective models treat agents like points in space whereas high-fidelity environment represent them as complex entities with mass, inertia and physical bounds.
- Scalability: the simulator should still be able to compete with other collective programming simulators in terms of nodes represented inside the experiments and their interactions.

Chapter 2

Background and State of the Art

To contextualize the contributions of this thesis, it is necessary to establish the theoretical foundations upon which it is built. This chapter explores the evolution of distributed systems toward collective intelligence and examines the formalisms of self-organizing frameworks. By evaluating the limitations of current simulators, this chapter identifies the technical 'reality gap' that this research aims to bridge, providing the necessary background to appreciate the integration of high-fidelity game engines into the decentralized coordination workflow.

2.1 Distributed Systems and Organizational Complexity

2.2 Self-Organizing Frameworks

2.2.1 Aggregate Computing

2.3 Simulation Landscape

2.3.1 Paradigms

2.3.2 The Reality Gap

2.3.3 Reealism vs. Scalability

2.4 Game Engines as Simulators

Chapter 3

Unity-Package-Template: Automated Unity Development Infrastructure

The mismatch between professional software engineering standards and the project-centric nature of the Unity Editor required the deliberate creation of this production-ready architecture. The environment's dependability is crucial in a research-focused setting such as the cro:CASComplex Adaptive Systems (CAS) simulator; if the underlying framework is non-deterministic or prone to human configuration errors, the validity of the data produced is compromized. The emphasis was moved from handling Unity's peculiar project structures to a modular, package-oriented approach by designing a 'template-first' methodology. by treating the simulator as a high-integrity library rather than a single executable, this methodology makes sure that the core functionality is independent of the particular Unity version or project parameters being used for visualization.

3.1 Requirements

Requirements for this project are split into separated categories. It aims at supporting both template users and developers.

3.1.1 Business Requirements

- The project shall provide a 'Zero-Friction' onboarding process by enabling a developer to move from cloning to a working development environment in a single step.

- The system must inject unique namespaces, names, and domains into the boilerplate code during initialization.

- The infrastructure shall automate repetitive tasks like dependency installation and secrets upload to GitHub.

- The project must treat the code as a distributable library rather than a standalone executable.

  - It must distinguish between Runtime and Editor content.
  - It must make no assumptions about the final playable build or entry point.

- The template must be self-evolving, allowing the infrastructure to be updated without breaking the projects that were instantiated from it.

3.1.2 Domain Requirements

Unity Ecosystem Domain

- The system shall enable package development with the ease of the Unity Editor.

- The repository structure must adhere to the cro:UPMUnity Package Manager (UPM) directory conventions [Uni26a].

- The system must handle the specific challenges of Unity asset version control, such as the merging of YAML-based files (scenes, prefabs, and metas).

DevOps Domain

- The infrastructure must guarantee deterministic versioning, where version increments are tied to the intent of the changes.
- The repository must enforce no development secrets, local licenses, or temporary editor files are leaked into the public source.
- The system must enable 'Policy as Code', making branching rules, quality gates, and release logic explicit and reviewable.

Trust Domain

- The system must provide authenticity ensuring that every release is signed and can be audited back to a specific commit.
- The infrastructure must support *living documentation*, where cro:APIApplication Programming Interface (API) references and usage guides stay synchronized with the evolving code.

3.1.3 Functional Requirements

- The system should allow a user to initialize a fully configured repository with a single command.
- The setup process must automatically configure project metadata across all internal configurations, including domain, company, package name, namespace, display name, description, and developer identity.
- The system should handle the automated injection of required secrets for CI/CD (Unity licenses, analysis tokens) during the initial bootstrap.
- The system must provide automated validation for code style and static analysis.
- The infrastructure should automatically execute tests across different categories (Runtime and Editor) in a headless environment.

- The system must prevent the integration of code that does not meet the defined quality gates or commit message standards.

3.1.4 Non-Functional Requirements

- The infrastructure must be executable on all three main OSes (i.e. Windows, Linux and MacOS).
- Setup failure must be 'loud' and diagnostic, providing clear error messages when prerequisites are missing.

3.2 Features and Tooling

To satisfy the requirements previously defined, a robust infrastructure was synthesized using a combination of industry-standard DevOps tools and bespoke automation logic.

3.2.1 Version Control System Management

The integrity of the codebase is maintained through a series of automated gates that prevent the accumulation of technical debt:

Husky To ensure a deterministic versioning history, Husky [hus26] is used to manage Git hooks. It enforces the Conventional Commits specification [con20], ensuring that every change intent is human-readable and machine-parsable. Moreover, on each commit, Husky performs fast local static checks including:

- automatic formatting of C# code to maintain stylistic consistency;
- validation of Unity-specific `.meta` and `.unity` (YAML) files to prevent asset corruption or missing references.

Semantic Release To close the loop between development and distribution, the project employs `semantic-release` [sem26]. This tool automates the entire package release workflow, including determining the next version number based on

the commit history, generating a `CHANGELOG.md`, and publishing release artifacts to GitHub without manual intervention.

3.2.2 Code Quality

SonarQube All source code is subjected to deep static analysis via SonarQube [son08]. This tool serves as a quality gate that evaluates code smells, security vulnerabilities, and cyclomatic complexity. By integrating this into the integration pipeline, the project ensures that only code meeting a specific 'Clean Code' threshold is merged into the stable branches.

cro:CIContinuous Integration (CI) The CI pipeline, powered by GitHub Actions, is designed to validate the project across a matrix of environments. It performs a multi-platform build (Windows, Linux, and macOS) to ensure cross-platform compatibility and executes both NUnit-based Runtime and Editor tests in a headless Unity environment.

Artifact Signing To ensure the authenticity and non-repudiation of the distributed package, the infrastructure includes an automated GPG signing stage. During the release process, the artifact is cryptographically signed using a private key generated during the initialization phase, allowing users to verify the integrity of the package.

3.2.3 Dependency Drift

Renovate To mitigate the risk of 'dependency rot', Renovate [ren26] is integrated into the repository. Unlike standard configurations, this project utilizes a custom regular expression manager to track and update dependencies within Unity's `package.json` and manifest files. This ensures that any project instantiated from the template always benefits from the latest security patches and engine features while providing a transparent review process through automated Pull Requests.

3.2.4 Developer Experience

Ensuring a flawless development experience is a major requirement for this project. Several tools and architectural patterns have been adopted to improve the developer's quality of life.

Unity Smart Merge Since Unity assets are stored in YAML, standard merge tools often fail to resolve conflicts in scenes or prefabs. The infrastructure automatically configures the *Unity YAML Merge* tool (SmartMerge) [Uni26b] within the local Git configuration, drastically reducing the friction of collaborative environment design.

DocFX To bridge the gap between source code and usable documentation, the project utilizes DocFX [doc26]. This tool functions as a static site generator that targets .NET assemblies. It extracts XML documentation comments (docstrings) directly from the C# source code to build a comprehensive, searchable API reference. By treating documentation as code, DocFX allows the infrastructure to host a specialized website that includes both conceptual manuals and technical API documentation. This ensures that the library's public surface is always accurately reflected in the documentation, preventing the common 'documentation lag' found in rapidly evolving projects.

Boot Command The project factory is encapsulated in a bespoke `init` script. This script orchestrates the transformation from a generic template to a specific project by:

- automatically renaming directories and updating Assembly Definitions (`.asmdef`) [uni20] with project-specific namespaces;
- injecting project-specific values (Domain, Company, Package Name, Namespace, Display Name, Description, Git username, and Git email) across all internal configurations;
- handling automated licensing based on user preferences;

- uploading repository secrets (e.g. GPG keys, Sonar tokens and Unity licenses) via the GitHub CLI [git26];
- installing npm and .NET dependencies;
- booting Unity in batch mode to generate the initial `.meta` files and installing Git hooks to ensure the environment is 'ready-to-edit' upon completion;
- opening the Unity Editor inside the sandbox for immediate development;
- creating the `develop` branch, applying automated branch and tag protection rules to the `main` and `develop` branches to enforce the quality gates from day one;
- committing all initialization changes and tagging that commit as `0.0.0`, which anchors the semantic versioning chain;
- removing the `.template` file and the `init` script itself, as they are no longer needed once the project has been bootstrapped.

Template Mode A key architectural challenge was that the template itself requires a CI pipeline for its own development and maintenance, while the projects instantiated from it require a fundamentally different one oriented towards package distribution. This duality is resolved through the presence of a single sentinel file: `.template`. When this file exists in the repository, the pipeline operates in *Template Mode*, executing the validation and release logic appropriate for the template infrastructure itself. Upon a successful `init` run, the file is deleted, and from that point forward the same pipeline switches to *Package Mode*, running the full package build, test, quality gate, signing, and release workflow. This design directly satisfies the self-evolving requirement: the template infrastructure can be continuously improved using its own automation, and any project instantiated from it inherits those improvements transparently via Renovate's automated dependency update pull requests.

Sandbox Pattern Because the Unity Editor cannot develop a package in complete isolation, the project implements a 'Sandbox' architecture. This consists of

a minimal Unity project within the repository that references the package via a local file path. This allows developers to use the full suite of Editor tools while ensuring the package itself remains clean and ready for external distribution via the UPM.

Chapter 4

Collektive×Unity: Designing a 3D Simulator for Collective Systems

This chapter face the core research project produced for this thesis: a simulator for 3D CAS.

4.1 Goal

The project goal is to bridge the Unity game engine with the aggregate computing library named Collektive.

This communication should be bidirectional, achieve high performance and enable huge customization.

4.2 Requirements

Requirements are split into separated categories.

4.2.1 Business Requirements

- The project should create a communication channel between the Collektive back-end and the Unity front-end.

- The project should extract environmental data from Unity node sensors and share them to the Collektive program.
- The integration must map the output of the Collektive aggregate program to Unity Actuators (e.g., changing position, color, or state of a GameObject).
- The system shall evaluate and implement a low-latency communication or bridge mechanism to minimize overhead between the JVM-based Collektive and the C#-based Unity environments.
- The integration should allow Collektive nodes to perceive Unity's colliders, rigidbodies and spatial triggers as first-class citizens.
- The integration layer should reamin agnostic to the specific CAS case study.

4.2.2 Domain Requirements

Simulator Domain

- The simulator should have customizable node sensors.
- The simulator should have customizable node actuators.
- The simulator should have customizable step duration (i.e. *delta time*).
- The simulator should be able to pause the simulation.
- The simulator should have a centered handling of randomization to enable reproducibility.
- The simulator should support addition and remotion of nodes in the simulation dynamically.
- The simulator should allow nodes to interacts at least with the following unity components:
  - rigid body
  - collider

- The simulator should support addition and remotion of neighbors dynamically.
- The simulator should support any kind of nighborhood discovery.

Communication Domain

- The communication should follow the reactive pattern (i.e. Collektive reacts to Unity's stimuli).
- The data exchanged should be agnostic from the underlying case study.
- Performance should be the driver for choosing the right technology.

Research Domain

- The system should prove the feasability of integrating game engines within CAS frameworks.

4.2.3 Functional Requirements

User Functional Requirements

- The user should treat a Unity scene as the simulation environment.
- The user should treat the Unity Editor as the simulator.
- The user should be able to add and remove nodes from the environment.
- The user should be able to create neighborhood discovery logic.
- The user should be able to inject any kind of collektive program inside the simulation.
- The user should be able to attach many different sensors and actuators to the same node.
- The user should be able to configure each node independently from the others.

System Functional Requirements

- The system should allow users to define custom sensors and acutators without modifying the core integration library.

- The system should allow users to define custom Collektive program without modifying the core integration library.

4.2.4 Non-Functional Requirements

- The system should maintain stable frame rate ($>$ 30 FPS) with at least 500 active collektive nodes in a low budget laptop (ryzen 7 5700U, 16GB DDR4, integrated GPU).

- The system should be implemented with the fastest technology found during exploration.

4.3 Architecture

The integration is designed as a modular bridge between Unity and Collektive, facilitating the bidirectional flow of information as illustrated in diagram 4.1.

Figure 4.1: Diagram showing the bidirectional communication bridge between Unity and Collektive.

4.3.1 Architecture Rationale

Before detailing the individual components, it is necessary to outline the core architecture decisions that ensure the system's scalability and flexibility.

From Application to Package The architectural design of this bridge was shaped by a shift in how the simulation environment is hosted. Early design iterations focused on developing a standalone simulation application. However, this approach led to a redundant replication of complex features already native to the Unity Editor, such as spatial partitioning, inspector-based configuration, and asset management.

To resolve this, the architecture was refactored from a monolithic application into a Unity Package. This shift leverages the Unity Editor as the primary configuration interface rather than a mere rendering target. By distributing the backend and bridge as a modular library, the user can utilize Unity's native tooling to define simulation parameters, modify environment geometry, and inspect node states in real-time. Consequently, the project functions as an extensible framework that transforms a standard Unity project into a specialized CAS simulator.

Delegated Neighborhood Discovery A critical architectural decision was the delegation of spatial logic to the Unity environment. In aggregate computing, determining network topology is fundamental. If this logic were handled in the Kotlin backend, every simulation tick would require an $O(n^2)$ complexity check to evaluate node proximity.

By moving discovery to Unity, the system exploits the engine's highly optimized physics engine and pre-baked spatial partitioning (e.g., grids or Octrees). This allows for efficient neighborhood calculations using native features like Colliders or Raycasting, ensuring the backend remains a pure logic engine that receives a pre-calculated adjacency list.

4.3.2 Component Implementation

The bridge architecture is decomposed into three distinct functional components that realize the design principles mentioned above.

Collektive Parser

To enable communication across a network or cro:IPCinter-process communication (IPC) layer, the Collektive API requires a robust serialization strategy. Two

primary design choices were made to facilitate this:

- the simulation runs on a single machine; thus, Collektive is configured as `InMemory` to minimize communication latency;
- the generic identifier in Collektive has been reified to an integer to ensure predictable serialization.

The primary challenge lay in the program's input and output, which are both generics. While maintaining these as generics is essential for flexible collective behavior, it complicates data exchange. Rather than the brittle approach of manually replicating data structures and custom serializers on both sides of the bridge, this implementation adopts an cro:IDLInterface Definition Language (IDL). By utilizing an IDL, the system decouples the data definition from the implementation language, allowing 'a program or object written in one language [to] communicate with another program written in an unknown language' [con].

Unity Parser

While the Collektive Parser handles the logic-side data mapping, the Unity Parser translates these definitions into the engine's entity-based structures. Unity operates on an cro:ECSEntity Component System (ECS) inspired architecture, where the state of any object is defined by its collection of attached components. To bridge this with the aggregate computing domain, the *Unity Parser* is implemented as a specialized component responsible for bidirectional data translation.

At the scene level, a central *Simulation Orchestrator* component serves as the gateway. This entity enhances a standard Unity scene into an active simulation environment by managing the communication channel's lifecycle. It interfaces directly with the *Core Bridge*, utilizing the IDL to serialize the environmental state and deserialize incoming commands from the Collektive back-end.

For individual nodes, the parser abstracts Unity's internal state into the language-agnostic format required by the aggregate program. The parser is responsible for neighborhood discovery. Rather than delegating spatial logic to the back-end, Unity identifies neighbors according to any arbitrary logic (e.g., physical proximity, line-of-sight or topological links) and provides them to the bridge as a collection

of identifier pairs. This ensures the system remains agnostic to the specific neighboring criteria, allowing the user to implement any discovery logic within the Unity environment.

Core Bridge

The Core Bridge represents the central link in the system, serving as the high-speed interface between the Unity C# environment and the Collektive Kotlin/JVM runtime. To satisfy the strict non-functional requirement of maintaining high frame rates with over 500 active nodes, the bridge is implemented as a cro:FFIForaign Function Interface (FFI) layer.

The choice of FFI over more traditional IPC methods was driven by the need to minimize serialization overhead and context-switching latency. By exposing Collektive's core logic as a native library, the bridge allows Unity to perform direct memory-to-memory communication. To ensure data consistency across the two environments, the bridge enforces a synchronous execution model. Each simulation 'tick' triggers a blocking call: the Unity engine pauses its internal loop while the bridge marshals the data and waits for the Collektive engine to complete its computation round.

The Bridge's role is strictly defined by three mechanical operations:

- Data marshalling: The process of transforming high-level language objects into a language-agnostic binary representation, as defined by the IDL schema.
- Function invocation: The execution of the foreign logic via the FFI, passing pointers to the marshalled data buffers.
- Data unmarshalling: The reconstruction of the binary data into the target language's native types, allowing the receiving environment to process the information.

A comprehensive performance comparison and a detailed evaluation of the trade-offs between FFI and Socket-based communication are provided in Chapter 7.1.

Chapter 5

Implementation of Collektive×Unity

This chapters goes into details about what produced in the Collektive×Unity project. Here it will be exposed the detailed design of the application.

5.1 Design

The design discussion will be divided into the 3 main components identified during the architecture phase: Collketive backend, Core Bridge and Unity frontend.

5.1.1 Collektive Backend

As stated in the architecture section (4.3) the communication across the bridge is synchronous. That means, each time Unity asks something to Collektive, it awaits its answer before restart doing its work. To enforce that architecture choice it has been created a central engine in charge of exposing the main capabilities of a CAS simulation. It can be seen as a facade pattern [GHJV94] in which the engine is the middleware from which the communication passes.

As shown in fig. 5.1 diagram this class should have methods to handle the whole simulation like `addNode`, `addConnection` and `step`. The `step` method in particular is the central point that should perform a cycle of the passed node. It is what moves the simulation one step ahead.

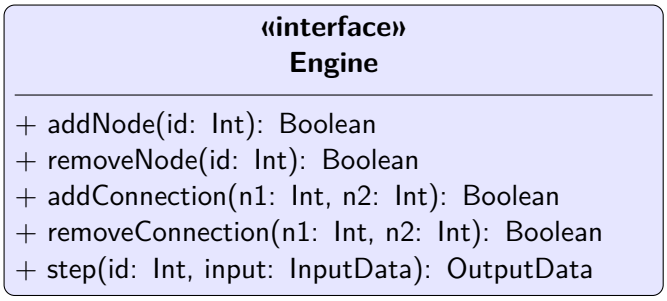

Figure 5.1: The Engine Facade.

```
1 fun Aggregate<Int>.entrypoint(sensorData: SensorData): ActuatorData
```

To follow the cro:SRPSingle Responsibility Principle (SRP) [Wik24] it has been created a helper component that is in charge of handling neighborhood and network in general for the simulation. This component is split into 2 objects `Network` and `NetworkManager`. The former is a stateless representation of the network visible from the perspective of a node (i.e. the implementation of the `Mailbox` in Collektive) while the latter is a master that knows the entire network and handles communication. The same `NetworkManager` is injected to each `Network` that will use it in order to send and receive messages from and to the neighborhood. The fig. 5.2 diagram shows the network API and their interaction.

From requirements 'the system should allow users to define custom Collektive program without modifying the core integration library'. To honor that requirement the core collektive program (from now on 'entrypoint') should be somehow injected into the engine, so that it can remain independent from the actual simulation it has being done. This inversion of control is achieved by injecting a lambda into the Engine constructor, as illustrated in Listing 5.1.1. This allows the Engine to remain agnostic of the specific collective logic being executed.

The entrypoint serves as the primary abstraction layer for the simulation designer (i.e. the simulator user), providing a restricted scope where collective behaviors are defined without exposure to the underlying engine orchestration. By isolating this logic into a standalone lambda, the framework ensures that the simulation's behavioral rules remain decoupled from the communication and synchronization mechanics of the backend.

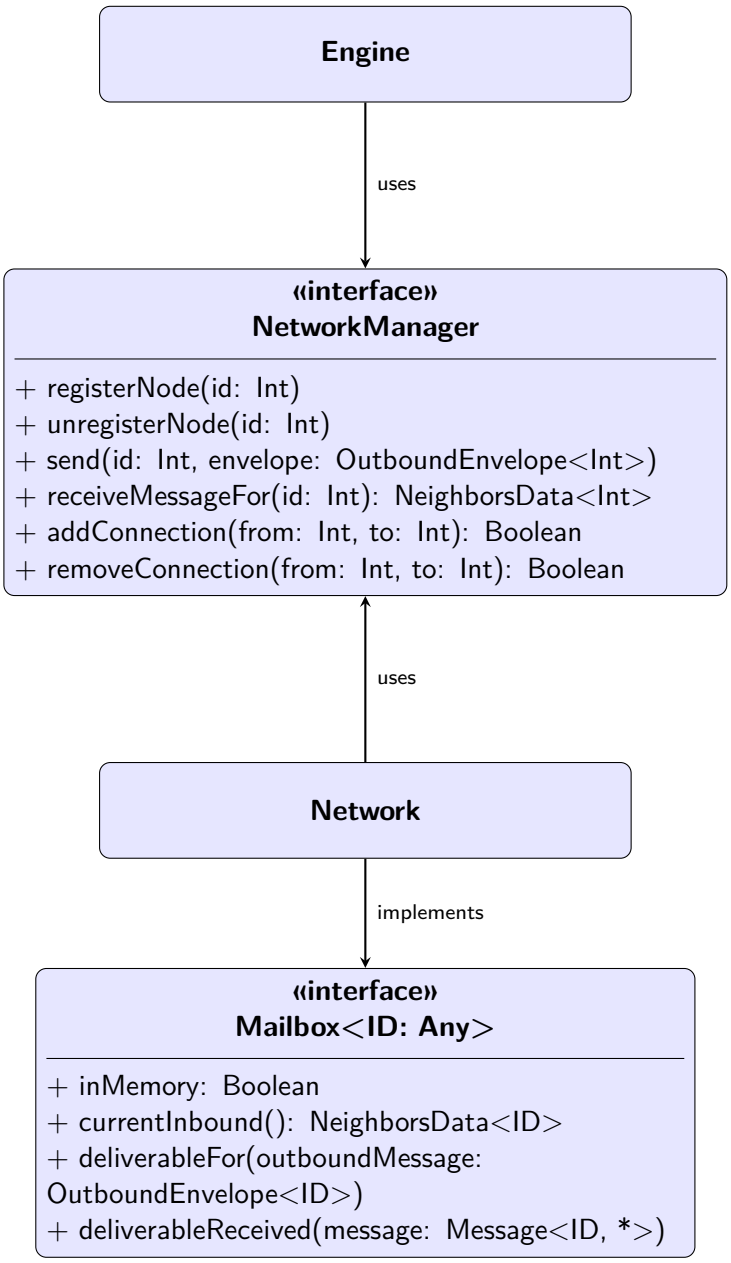

Figure 5.2: How networking is handled in the Collektive backend.

It is important to note that the `SensorData` and `ActuatorData` types utilized in the entrypoint signature represent the boundary between Collektive and Unity. To maintain strict type-safety across this boundary, these structures are centralized in the *Core Bridge*, acting as the common interface through which simulation state and environmental sensors are exchanged.

5.1.2 Core Bridge

The Core Bridge serves as the critical 'glue code' that enables communication between the Collektive and Unity environments. As detailed in Section 4.3, it leverages an IDL to marshal domain-specific data structures into low-level byte arrays. These are transferred via a FFI to the opposing context, where they are reconstructed into native data structures.

The IDL selected for this implementation is Protocol Buffers. While FlatBuffers was initially considered for its superior 'zero-copy' performance [vO14], it was ultimately discarded due to compatibility constraints with cro:KMPKotlin MultiPlatform (KMP) projects.

Protocol Buffers As stated in the official documentation, Protocol Buffers provide a 'language-neutral, platform-neutral, extensible mechanism for serializing structured data [...] You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data' [Goo24]. This system utilizes a specialized compiler (`protoc`) that processes a `.proto` definition file to generate the required source code for each target environment (e.g., `.kt` for Kotlin and `.cs` for C#).

The adoption of an IDL extends beyond mere serialization efficiency; it enforces the cro:DRYDon't Repeat Yourself (DRY) principle by providing a single source of truth for data definitions. This decoupling ensures that the data schema remains synchronized across both the cro:JVMJava Virtual Machine (JVM) and the Unity engine, reducing the risk of structural mismatches during the development of complex collective behaviors.

Usage The responsibility for defining shared data structures is delegated to the simulation designer. Users must define the following two core structures within the IDL schema:

- `SensorData`: the information captured by Unity components from the 3D environment to be sent to the Collektive backend;
- `ActuatorData`: The resultant data produced by the Collektive program, which is returned to Unity to trigger physical or logical actions.

The decision to delegate these definitions to the user is intentional and ensures the framework remains domain-agnostic. In a monolithic environment, this flexibility would typically be achieved through generics or polymorphism. However, since data must cross a language barrier where native generic types cannot be shared, the IDL serves as a functional substitute. By requiring the user to define these structures, the bridge maintains its genericity, allowing it to support any simulation type without requiring changes to the underlying bridge implementation.

Data Transmission Once the user-defined data structures are generated, the framework manages the lifecycle of their transmission across the bridge. When a simulation 'tick' occurs, the Unity engine serializes the `SensorData` into a compact binary format using the Protocol Buffers library. Lastly, the Core Bridge passes a pointer to the underlying byte array through the FFI channel.

On the receiving end, the Collektive backend reads from this shared memory location and reconstructs the data into native Kotlin objects. This process ensures that while the user works with high-level, type-safe classes in both C# and Kotlin, the actual communication remains a memory-bound operation.

5.1.3 Unity Frontend

The frontend serves as the final integration layer, consolidating the previously described components into a unified simulation environment. To align with Unity's

Component-Based architecture, the system utilizes specialized `MonoBehaviour` entities to bridge engine-level events with aggregate logic. Central to this coordination is the `SimulationManager`.

Acting as the functional counterpart to the Collektive `Engine`, the `SimulationManager` is a singleton-like component attached to a unique orchestrator object within the scene. Its role is twofold:

- environment configuration: it initializes the unity physics engine and global simulation parameters to ensure deterministic execution;
- interface bridging: it reflects the high-level facade of Collektive backend `Engine`. By interfacing with a static service layer that utilizes the cro:P/InvokePlatform Invoke (P/Invoke) mechanism [Mic24], the manager exposes native library capabilities to the Unity domain.

As illustrated in diagram 5.3, this creates a continuous communication pipeline where the `SimulationManager` orchestrates the flow of data from the Unity scene, through the binary serialization layer, and finally into the native JVM execution context.

Along with the `SimulationManager` the project introduces also another singleton: the `LinkManager`. `LinkManager` is a visualization-oriented singleton that maintains and renders the current network links (when enabled), decoupling debugging/inspection concerns from simulation logic.

Aggregate programs depend on neighbor exchange, but the definition of *who is a neighbor* is inherently situated and environment-dependent. In the project, neighborhood construction is handled on the Unity side by creating and removing directed or bidirectional links through the `SimulationManager` API. This enables multiple strategies, ranging from geometric proximity to occlusion-aware connectivity or application-specific policies. A common and efficient pattern leverages Unity's physics system and trigger colliders. For instance, a connection rule component that automatically subscribes/unsubscribes links when other nodes enter/exit a proximity volume can be implemented by annotating the implementing class with an attribute, whose `OnTriggerEnter` and `OnTriggerExit` methods call the appropriate link-management API to maintain the neighborhood structure.

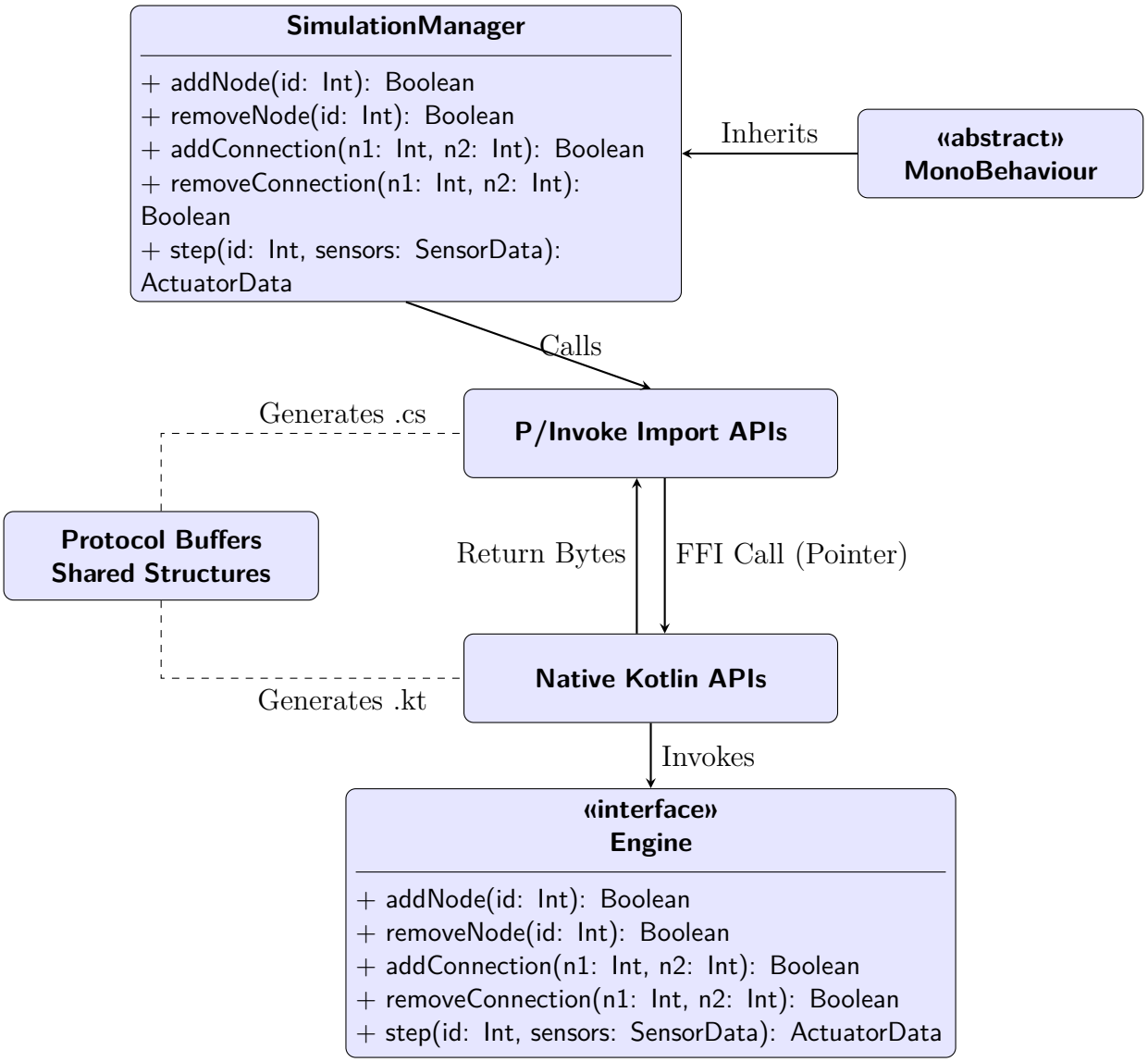

Figure 5.3: Communication channel from the Unity domain down to the Collektive boundary.

By delegating neighborhood dynamics to Unity, the integration can directly reuse engine-level facilities (collisions, triggers, layers, and spatial partitioning), while keeping the Collektive runtime agnostic of the specific spatial model.

The final piece in the Unity frontend is the `AbstractNode`. `AbstractNode` marks a game object as a Collektive device. It defines the extension points that are scenario-specific: how sensor data is sampled from the Unity world, and how actuator commands are applied. Concrete scenarios are implemented by subclassing `AbstractNode` and providing the domain logic for sensing and actuation, while reusing the integration-provided communication and execution machinery.

A final overview of the frontend is presented in fig. 5.4

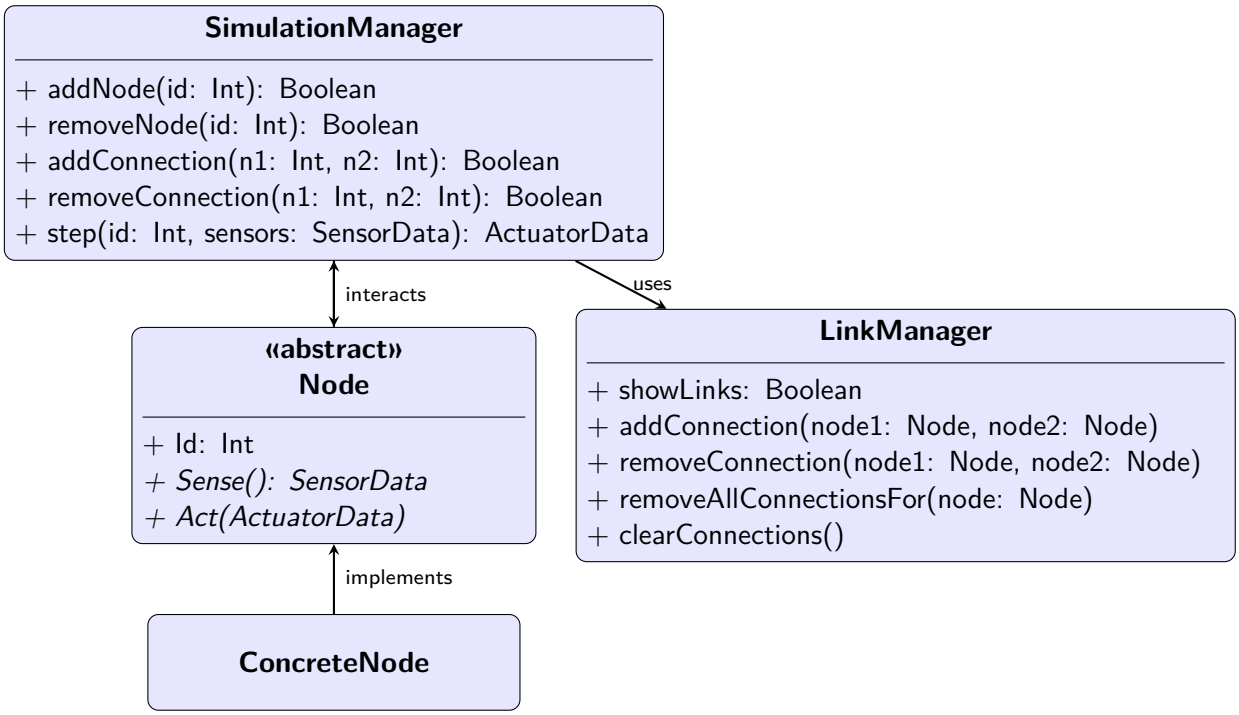

Figure 5.4: General overview of the frontend components.

5.2 Implementation Details

In this section it will be explained in more details certain parts of the project that have intresenting insights.

5.2.1 Unity assets

Unity provides a robust asset system centered around *Prefabs*, which were instrumental in this project's development. According to the official documentation, the Prefab system 'allows you to create, configure, and store a GameObject complete with all its components, property values, and child GameObjects as a reusable asset' [Uni]. By leveraging this system, a `SimulationManager` Prefab was developed. This asset encapsulates both the `SimulationManager` and `LinkManager` components; consequently, a standard Unity scene can be converted into a simulation environment simply by instantiating this single Prefab.

5.2.2 Design Patterns in Unity

Implementing traditional design patterns within the Unity ecosystem presents unique challenges. Because Unity utilizes a Component-Based Architecture rather than a strict object-oriented hierarchy, maintaining high-quality, decoupled code is not always intuitive. For this project, the `SimulationManager` and `LinkManager` were architected as Singletons to centralize simulation logic.

In general software engineering, the Singleton pattern is intended to 'ensure [classes] only have one instance, provide easy access to that instance, and control their instantiation' [Wik25]. However, in Unity, a script's lifecycle is inextricably linked to the `GameObject` to which it is attached. To reconcile the Singleton pattern with Unity's scene-based execution, the `SingletonBehaviour` class was implemented, as illustrated in listing 5.1.

The implementation leverages the Unity engine's APIs to enforce instance uniqueness across the execution lifecycle:

- the `Object.FindObjectOfType<T>` method is employed to query all active `MonoBehaviour` instances within the current scene hierarchy, returning the existing instance or `null` if none is present;
- the `AddComponent<T>` method is used to attach the component to a programmatically instantiated `GameObject` in the event that no prior instance is detected;

Listing 5.1: Implementation of the Singleton pattern in Unity.

```
public abstract class SingletonBehaviour<T> : MonoBehaviour
    where T : MonoBehaviour
{
    private static T _instance;
    private static readonly object _lock = new object();

    /// <summary>
    /// The singleton instance.
    /// </summary>
    public static T Instance
    {
        get
        {
            lock (_lock)
            {
                if (_instance == null)
                {
                    _instance = (T)Object.FindAnyObjectByType(typeof(T));
                    if (_instance == null)
                    {
                        var singleton = new GameObject();
                        _instance = singleton.AddComponent<T>();
                        singleton.name = "(singleton) " + typeof(T).ToString();
                        DontDestroyOnLoad(singleton);
                    }
                }
                return _instance;
            }
        }
    }
}
```

- the `DontDestroyOnLoad` method is invoked to ensure the persistence of the manager across scene transitions, preventing the engine from deallocating the object during level loads.

5.2.3 Unity Editor Customization

The Unity environment offers a robust framework for extending the Editor's native capabilities through user-defined scripts. This project leverages this extensibility for two primary objectives:

- the automation of repetitive build and configuration tasks;
- the enhancement of runtime observability and debugging.

Workflow Automation During the development phase, the integration of the backend required frequent recompilation of native libraries and the relocation of the resulting `.so` binaries into specific Unity-accessible directories. Additionally, updates to the communication protocol necessitated the regeneration of source code from `.proto` definitions. To streamline these processes, two custom menu utilities were implemented within the Editor's global navigation bar:

- Native Rebuild: Accessible via `Tools > Native > Rebuild backend`, this utility automates the Collektive backend build process and ensures the binary is correctly placed for linking.
- Protocol Generation: The `Tools > Proto > Generate` option triggers the Protocol Buffer compiler to process `.proto` files and output the generated classes directly into the project's script folder.

Runtime Debugging and Observability Because Unity employs a proprietary internal console for logging, monitoring the internal state of the native backend during execution can be cumbersome. To mitigate this, a custom `ReadOnly` attribute was developed.

When applied to a field already marked with Unity's `SerializeField` attribute, this custom decorator ensures the value is rendered in the *Inspector* as a read-only field. By binding these fields to data returned from the backend, the *Inspector* serves as a real-time monitoring dashboard. This allows for the observation of backend state changes directly within the Editor UI without the risk of manual accidental modification by the user. A screenshot of this can be seen at fig. 5.5.

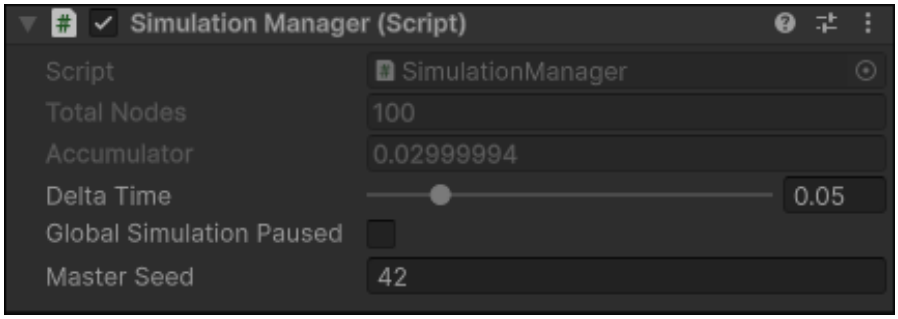

Figure 5.5: Screenshot that shows how readonly properties appear in the Unity Editor *Inspector* tab. In this image *Total Nodes* and *Accumulator* are both readonly.

Chapter 6

Case Study: Environment-aware Gradient Ascent

Chapter 7

Results

7.1 Comparison with Socket-based Communication

Chapter 8

Conclusions and Future Work

Bibliography

[con] Wikipedia contributors. Interface description language.

[con20] Conventional commits 1.0.0, 2020. Accessed: 2026-02-23.

[doc26] Docfx documentation, 2026. Accessed: 2026-02-23.

[GHJV94] Erich Gamma, Richard F. Helm, Ralph E. Johnson, and John Vlissides. *Design patterns: elements of reusable Object-Oriented software*. 1994.

[git26] Github cli — take github to the command line, 2026. Accessed: 2026-02-23.

[Goo24] Google LLC. *Protocol Buffers: Google's Data Interchange Format*, 2024. Accessed: 2026-02-17.

[hus26] Husky — modern native git hooks, 2026. Accessed: 2026-02-23.

[Mic24] Microsoft. Platform invoke (p/invoke), 2024. Accessed: 2026-02-17.

[ren26] Renovate docs, 2026. Accessed: 2026-02-23.

[sem26] semantic-release documentation, 2026. Accessed: 2026-02-23.

[son08] Sonarsource – better code & better software, 2008. Accessed: 2026-02-23.

[Uni] Unity Technologies. Prefabs - unity manual.

[uni20] Script compilation and assembly definition files, 2020. Accessed: 2026-02-23.

[Uni26a] Unity Technologies. *Differences between package types - Unity Manual (Version 6000.3)*. Unity Technologies, 2026. Accessed: 2026-02-23.

[Uni26b] Unity Technologies. *SmartMerge - Unity Manual (Version 6000.3)*. Unity Technologies, 2026. Accessed: 2026-02-23.

[vO14] Wouter van Oortmerssen. Flatbuffers: Memory efficient serialization library. Technical report, Google, 2014. Accessed: 2026-02-17.

[Wik24] Wikipedia contributors. Single-responsibility principle — Wikipedia, the free encyclopedia, 2024. [Online; accessed 17-February-2026].

[Wik25] Wikipedia contributors. Singleton pattern — Wikipedia, the free encyclopedia, 2025. Last edited 17 October 2025; [Online; accessed 21-February-2026].

Acknowledgements

Optional. Max 1 page.