Miles Watson

# TCP ex Machina

Christ's College
University of Cambridge

This dissertation is submitted as part of the requirements for

*Part II of the Computer Science Tripos*

May 2024

# Declaration

I, Miles Watson of Christ's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: *Miles Watson*

Date: *May 10th, 2024*

# Proforma

| | |
|---|---|
| Candidate Number: | **2411D** |
| Project Title: | **TCP ex Machina** |
| Examination: | **Computer Science Tripos – Part II, 2024** |
| Word Count: | **11,768** [1] |
| Code Line Count: | **6,372** [2] |
| Project Originator: | **Prof. Jon Crowcroft** |
| Project Supervisor: | **Prof. Jon Crowcroft** |

## Original Aims of the Project

Computer-generated congestion control (CGCC) is the practice of using computers to generate congestion control algorithms (CCAs). The original aims of this project were to build an extensible framework for CGCC using Rust, to implement the "Remy" algorithm using the framework, and to evaluate the learnt CCAs using a popular network simulator. As an extension, the project also sought to implement and evaluate an algorithm based around reinforcement learning (RL).

## Work Completed

The project was a success, fulfilling all the core criteria and the most ambitious extension. We started by creating a formal definition for the problem of learning congestion control, and used it to inspire the design of FlowForge – a practical framework for CGCC. Using the framework, we implemented Remy, and used a popular network simulator to show the produced CCAs are comparable to those of the original implementation. We also designed and implemented Remyr – a novel variant of Remy based around RL – and showed that it produces CCAs more performant and understandable than those of the Remy.

## Special Difficulties

[Redacted]

[1]Pasted into `https://wordcounter.net` due to an ongoing underestimation issue with `typst-wordometer`.

[2]Lines of Rust code in the core repository. Excludes code used for evaluation. Counted using `tokei`.

# Contents

# 1 Introduction

This dissertation explores **computer-generated congestion control (CGCC)** – the process of using computers to generate *congestion control algorithms*. Specifically, we seek to **build and evaluate a framework for learning congestion control**. In this section, we give an overview of the field, and briefly summarise the aims of the project.

## §1.1 Motivation

**Challenges in congestion control.** **Congestion-control algorithms (CCAs)** are algorithms responsible for regulating the entry of traffic onto a network, in order to prevent *congestive collapse* – a state where traffic demand is high, but little useful throughput is available [Nagle, 1995]. Traditionally, CCAs are *human-generated* – human researchers make assumptions about the networks that the algorithms will be used on, and then implicitly encode these assumptions in the produced CCA.

**TCP ex Machina.** In their 2013 paper, Winstein and Balakrishnan proposed a different paradigm for end-to-end congestion control that they called *computer-generated congestion control*. In this paradigm, an algorithm takes explicitly encoded assumptions about the network as input and outputs a CCA optimised for the given assumptions. They also present **Remy**, an example of such an algorithm based on *decision trees*, which can produce CCAs that outperform traditional approaches.

**A modern approach.** **Reinforcement learning (RL)** is a field of machine learning in which agents interact with an environment and learn to maximise reward. Inspired by Remy, recent approaches such as *Aurora* [Jay et al., 2019] and *RL-CC* [Tessler et al., 2022] have found success by applying RL to congestion control.

**Existing solutions.** Existing programs to learn congestion control primarily use Python, C++, or a mixture of both. Python is the industry standard for machine learning, but its performance makes it a poor choice for the "hot path" of a network simulator. C++ is much faster, but suffers from poor tooling and makes it easy for programmers to cause undefined behaviour.[3] Overall, existing tools focus primarily on *research* rather than the applications to *production use in industry*. In addition, they are often either highly specialised or overly generalised, making extensibility difficult.

**Rust: performant defined behaviour.** Rust is a modern programming language with an emphasis on performance, type safety, and concurrency. It uses a number of compile time techniques to guarantee that code not marked `unsafe` is free from data races, segfaults, double frees, and other classes of error common in C++ [Jung et al., 2017]. This reduced likelihood

[3]Poor tooling is in comparison to Rust's default tools, e.g. `rustup`, `rustc`, `rustfmt`, `cargo`, `clippy`, `rustdoc`. The author found an example of such undefined behaviour in the Remy repository.

of introducing subtle bugs, combined with the excellent standardised tooling, makes projects using it conducive to new contributors.

## §1.2 Project aims

Fundamentally, this project aims to

- **build a framework for learning congestion control**,
- implement **Remy** using the framework, and
- explore **reinforcement learning** as a method for learning congestion control.

As previously mentioned, existing solutions focus on *researchers* as the target demographic. Because of this, they usually have little consideration for performance, developer experience, or reliability – important factors to consider when developing solutions for industry. Instead of taking this approach, we consider the target demographic to be *software engineers* employed in the industry. Through this change of focus, we intend to support high-performance implementations of methods for learning congestion control, and to facilitate easier integration of CGCC into existing network infrastructure.

Potential use cases for the framework include:

- one-time training of a CCA for use in a satellite mesh network
- automatic training of new CCAs to adapt to changing traffic conditions in a datacenter
- researching methods of learning congestion control where inference would be impractical to perform in Python

# 2 Preparation

Before diving into the technical details, we must first establish some foundations of the project (such as the requirements and development methodology), and then cover some of the theoretical background that underpins the implementation.

## §2.1 Project foundations

In this section, we consider the starting point that the rest of the project builds upon, provide a breakdown of the requirements that the project should aim to satisfy, and review some of the software engineering tools and techniques used throughout the project.

### §2.1.1 Starting point

**Concepts.** Whilst the author had limited prior experience with Rust and a basic understanding of machine learning from Part IA and IB courses, they had no experience with reinforcement learning, the use of ML libraries, or the concept of computer generated congestion control.

**Tools and code.** To reduce project risk, the author made the decision to build directly upon the modified `ns-2` (a popular network simulator) codebase used by Winstein and Balakrishnan [2013] to evaluate the learned CCAs. The rest of the framework was written from scratch, with the original Remy implementation being used for occasional reference in cases where their paper is ambiguous.

### §2.1.2 Requirements analysis

We now present the criteria that will be used for evaluating the project. These are fundamentally the same as those in the original project proposal (Appendix F), but have been refined by adding additional requirements that consider the target demographic (software engineers) and the broader impact of the work.

#### Core aims

As given by the original project proposal, the core aim of the project is to build an extensible framework for learning congestion control, and to use it to implement the "Remy" training algorithm. In particular, we aim

a) to **build a framework** for learning congestion control (Section 3.1) that
   - contains a **flexible simulator** for high-performance emulation of real-world network behaviours (Section 3.2),

  - implements the **Remy algorithm** for learning congestion control (Section 3.3), and
  - is written primarily in the **Rust programming language**;

b) to **evaluate the performance** of the learned Remy CCAs (Section 4.1) against both
  - CCAs trained using the original Remy implementation, and
  - other traditional congestion control algorithms; and

c) to **evaluate the utility** of the framework for research and industry through qualitative analysis (Section 4.3), by considering
  - **performance** of the core framework,
  - **extensibility** given the provided abstractions,
  - **reproducibility** of results, and
  - **developer experience** when working with the framework.

The proposed extensions focus on using the framework to further explore methods of learning congestion control. Whilst we provide the capability for easily investigating two of them (genetic algorithms and more varied network architectures), this dissertation focuses on the most ambitious extension – exploring reinforcement learning as a method of learning congestion control.

**Extension aims**

We seek

a) to implement a training mode that uses **reinforcement learning** to learn CCAs (Section 3.4),

b) to **evaluate the performance** of the learned CCAs against other congestion control algorithms (Section 4.1), and

c) to **compare the approach** against that of our Remy implementation (Section 4.2).

To best fulfill the last requirement, we choose to limit ourselves to creating Remy-like CCAs. This allows us to provide a fairer comparison between the two methods of learning.

### §2.1.3 Software engineering techniques

**Development methodology.** Due to the contrast between the well-defined nature of the core aims and the unfamiliar domain of the extension, the author chose to divide the project development into two distinct phases. The first phase used the **waterfall methodology** to efficiently fulfill the core criteria within a set timeframe, whilst the second phase used the **spiral model** [Boehm, 1988] to iteratively develop and refine the extension.

**Testing strategy.** The project makes frequent use of Rust's built-in unit testing functionality to ensure that individual components function as expected. A notable example of this testing is the use of **round-trip testing** [Shilliday et al., 2010] to ensure compatibility with the original Remy DNA format. We ensure these unit tests are run often by using a GitHub Actions pipeline to automatically run tests on each push to the repository. However, not all behaviours in the project can be easily captured with unit tests – some are simply too complex to run through by hand. To work around this, we take inspiration from the primary testing strategy used by `ns-2` and employ **snapshot testing** using the `insta` testing tool [Ronacher, 2019]. This involves comparing machine output against manually-verified output from a previous run. By carefully ensuring determinacy of the program for a given seed, this allows us to implement characterisation tests for both network simulation and training processes.

**Code style and linting.** We use the built-in `rustfmt` tool to auto-format our code on save, alongside the `clippy` linter with `warn(clippy::pedantic, clippy::nursery)` to ensure best practices.

**Languages and libraries.** The project involved working with a variety of languages and libraries. Understanding existing SOTA RL implementations required familiarity with `PyTorch` [Paszke et al., 2017], whilst the actual implementation used `dfdx` – a Rust library for shape-checked deep learning [Lowman, 2022]. This library was chosen as it provides a similar level of abstraction to `PyTorch`, but without the need to make expensive calls into `Python`. Although the core framework was written in Rust, interfacing with `ns-2` required altering the source code and working with small amounts of `C++`, `Tcl`, `Perl` and `Make`. Figures in the evaluation section were produced using `Python`.

**Software licensing.** Although not yet ready for public release, the author intends to eventually open-source the project as `Apache-2.0 OR MIT`. We used `cargo-license` [Aslan, 2016] to aggregate the licenses for all 130+ direct and transitive dependencies, and determined that the project and its dependents must comply with `Apache-2.0 AND MIT AND BSD-3-Clause AND Unicode-DFS-2016`. Since no dependencies are modified or included directly in the repository, the primary impact is that copies of the dependency licenses (and notices if appropriate) must be included if distributing compiled versions of the framework.

**Software tools.** At various points during the development, `cargo-flamegraph` [Ferrous Systems, 2019] was used to profile the framework and guide performance optimisations. `GDB` was also used to debug segfaults when working with `ns-2`.

**Computational resources.** The project was largely developed on an M1 Macbook Air, with training performed on a Windows desktop computer (Ryzen 2600x with 6C12T @ 3.6GHz, 16GB memory, RTX 2060 with 6GB GDDR6). Evaluation was performed in an Ubuntu VM on the same desktop.

**Version control and backup.** Code was committed frequently to a GitHub repository (over 260 commits), with branches used for experimentation and larger changes. The repository was frequently backed up to both a cloud drive and an external HDD to protect against hardware or software failure.

## §2.2 Technical preliminaries

In this section, we provide some background on reinforcement learning, and attempt to formalise the problem of learning congestion control. We assume a working knowledge of topics covered in IA and IB.

### §2.2.1 Reinforcement learning

As mentioned in the introduction, reinforcement learning (RL) is a field of machine learning in which agents interact with an environment and learn to maximise reward [Achiam, 2018]. This chapter introduces a number of different environment models and algorithms designed to solve them – these are conveniently summarised in Appendix A.

**POMDPs**

We define a **policy** as a stochastic function $\pi$ from observations to actions. $\pi_\theta$ is a policy parameterised by $\theta$. A policy interacts with an environment by repeatedly:

- making an observation $o$ from some hidden state $s$
- taking an action $a$ with probability $\pi(a|o)$
- receiving a reward $r$ for that action

A **trajectory** is a sequence of $(s,o,a,r)$ tuples, one for each timestep in a contiguous period of time.

**Definition 1**

The **discounted return** of a trajectory $\tau$ is

$$R(\tau) = \sum_{n=0}^{|\tau|-1} \gamma^n r_n \tag{2.1}$$

where $r_n$ is the reward at step $n$, and $\gamma \in (0,1]$ is a discount factor.

Typically, environments in RL are described as discrete-time **Partially Observable Markov Decision Processes (POMDPs)**. The goal of a discrete-time MDP is to find a policy that maximises the expected future discounted return from the initial state $s_0$, denoted $V^\pi(s_0)$.

**Definition 2**

The **on-policy value** of a state $s$ for a policy $\pi$ is

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau) \mid s_0 = s] \tag{2.2}$$

where $\tau$ are infinite-length trajectories drawn from $\pi$ and $s_0$ is the initial state.

An important concept is the *advantage* of an action, which describes how much better taking an arbitrary action $a$ is in the current state than following the policy.

**Definition 3**

The **advantage** of an action $a$ in a state $s$ for a policy $\pi$ is

$$A^{\pi}(s,a) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau) \mid s_0 = s, a_0 = a] - V^{\pi}(s) \tag{2.3}$$

where $r_n$ is the reward at step $n$, $s_0$ and $a_0$ are the initial state and action respectively, and $\gamma \in (0,1]$ is a discount factor.

**Policy gradient algorithms**

One popular approach to RL is the use of policy gradient algorithms. These methods work by iteratively nudging the policy towards actions with higher advantages, and away from those with lower advantages. A requirement for this approach is that the policy must be *differentiable*. This means that the policy should be described in terms of some parameters $\boldsymbol{\theta}$, and that it should be possible to determine the gradient of an action with respect to $\boldsymbol{\theta}$.

A commonly used class of differentiable functions are **multi-layer perceptrons (MLPs)**. Multilayer perceptrons are formed by the composition of functions $f = f_1 \circ \ldots \circ f_n$ where each $f_i(x)$ is of the form $\Phi_i(\boldsymbol{W}_i x + \boldsymbol{b}_i)$ for a matrix of weights $\boldsymbol{W}_i$, a vector of biases $\boldsymbol{b}_i$, and a component-wise non-linear activation function $\Phi_i$.

An example of a policy gradient algorithm is **proximal policy optimisation (PPO)** [Schulman et al., 2017], which "clips" the objective to prevent destructively large policy updates when performing multiple iterations of optimisation. PPO is commonly implemented as an *actor-critic* algorithm. When implemented this way, the training process learns two functions – the policy $\boldsymbol{\pi}_\theta$ (known as the actor), and an estimator for the value function $\boldsymbol{v}_\theta$ (known as the critic). Both functions take the current observation $\boldsymbol{o}$ as input, with the actor producing an action $a$ and the value function producing $\hat{V}$, an estimate for $V^{\pi}(s)$.

## §2.2.2 From RL to MARL

RL and its popular POMDP model deal with the actions of a single agent interacting with the environment. In congestion control, however, we have multiple agents, each potentially with its own state,[4] operating asynchronously.

**Multi-agent reinforcement learning (MARL)** is an extension to reinforcement learning that deals with environments containing multiple agents. One popular model used by problems in this space is **decentralised POMDP (Dec-POMDP)** [Oliehoek and Amato, 2016], which models multiple independent agents making observations and taking actions each step. This model is commonly applied to end-to-end congestion control [Hemmati et al., 2015, Sivaraman et al., 2014].

[4]If learning a stateful policy, such as a RNN.

One approach to learning Dec-POMDPs is the *centralised training, decentralised execution (CTDE)* framework. In CTDE, the training process has access to information about the global state, but the trained agents must function with only local observations. One particular application of CTDE is a variant of PPO called **multi-agent PPO (MAPPO)**, which has showed "surprising success" in co-operative multi-agent tasks [Yu et al., 2024]. MAPPO alters the actor-critic model by allowing the critic access to the global state, whilst the actor is only given local observations.[5]

However, Dec-POMDPs have a limitation – all agents must step synchronously together. This generally limits CC approaches to those which *proactively* adjust the rate at fixed intervals, rather than those which *react* to events. In their 2022 paper, Liang et al. define an **asynchronous decentralised POMDP (AD-POMDP)** as follows:

**Definition 4**

**AD-POMDP** is defined as a tuple $\mathcal{M} = \langle \mathcal{N}, \mathcal{S}, \mathcal{O}, \mathcal{A}, \Omega, U, P, R, \gamma \rangle$ where

- $\mathcal{N} \equiv \{1, 2, ..., M\}$ is a finite set of $M$ agents
- $\mathcal{S}$ is the global state space of all agents
- $\mathcal{O} \equiv \langle \mathcal{O}^1, ..., \mathcal{O}^M \rangle$ are the observation spaces for each agent
- $\mathcal{A} \equiv \langle \mathcal{A}^1, ..., \mathcal{A}^M \rangle$ are the action spaces for each agent
- $\gamma \in (0, 1]$ is the discount factor

As agents in AD-POMDP make decisions asynchronously, at time slot $t_i$ there is a set of available agents $\mathcal{N}' \subseteq \mathcal{N}$. Only the agents in $\mathcal{N}'$ need to make decisions at time slot $t_i$. $\mathcal{N}'$ is obtained from function $U : \mathcal{S} \to \mathcal{P}(\mathcal{N})$, given the global state $s \in \mathcal{S}$. Correspondingly, the observation vector $\boldsymbol{o}$ of agents over $\mathcal{N}'$ can be obtained from function $\Omega : \mathcal{S} \times \mathcal{N}' \to \mathcal{O}^{|\mathcal{N}'|}$. Agents $j$ in $\mathcal{N}'$ select an action $a_j \in \mathcal{A}^j$ according to their own observation $o_j$, forming a joint action vector $\boldsymbol{a} \in \mathcal{A}^{|\mathcal{N}'|}$. The joint action execution of these agents results in a transition to the next global state $s' \in \mathcal{S}$ through (probabilistic) transition function $P : \mathcal{S} \times \mathcal{A}^{|\mathcal{N}'|} \to \mathcal{S}$. $\boldsymbol{r}$ is a reward vector from the reward function $R : \mathcal{S} \times \mathcal{A}^{|\mathcal{N}'|} \to \mathbb{R}^{|\mathcal{N}'|}$.

For our purposes, we define a co-operative variant **AD'-POMDP** where $R : \mathcal{S} \times \mathcal{A}^{|\mathcal{N}'|} \to \mathbb{R}$ is a global reward function. This is a superset of Dec-POMDP, allowing simpler reuse of existing algorithms for the model.

We can convert an AD'-POMDP into a **continuous AD'-POMDP** by approximating a unit of continuous time as some number of steps $g$ (where $g \to \infty$) and dividing the reward at each step by $g$. The discounted return is then given by

$$R(\tau) = \int_{t=0}^{\infty} e^{-\alpha t} r_t \, \mathrm{d}t \tag{2.4}$$

for discount rate $\alpha = \gamma^g$. This is the standard formula for expected discounted return used in continuous-time MDPs [Miller, 1968]. We note that the "asynchronous" idea in this case

[5] As with regular PPO, the critic is only used during training, and not during execution.

is similar to that of **semi-MDPs**, which approximate continuous MDPs as discrete MDPs where time "jumps" forward by variable amounts between actions [Bradtke and Duff, 1994].

### §2.2.3 Formalising CGCC

Providing a formal definition of computer-generated congestion control has two purposes. Firstly, it provides us with a language that allows us to more concisely and precisely communicate ideas. Secondly, it provides us with the foundation from which we will build a practical framework for CGCC.

#### Requirements

We first aim to construct a formal model of congestion control, and then build upon that to construct a formal definition for the problem of learning congestion control.

Our definition should support:

- discrete events that happen in continuous time
- uniform, end-to-end TCP CCAs, as in Winstein and Balakrishnan [2013]
- arbitrary probabilistic packet delay, duplication, loss, corruption, and reordering
- arbitrary probabilistic traffic patterns
- optimisation of CCAs for *Quality of Service* (as opposed to *Quality of Experience*)

#### Strategy

We approach this problem by decomposing the "essence" of a **Network**$_L$ into some core components:

- a **Sender**$_L$, with two instances per TCP connection, responsible for converting between byte and packet streams
- an **Interconnect-Behaviour**, responsible for dictating how packets are moved between senders
- a **Traffic-Behaviour**, responsible for producing and consuming one bytestream per sender

We separately define a **CCA**$_L$, with one instance per sender instance, responsible for controlling the congestion window with maximum size $L$.

We present a sketch for formal representations for these components, and show how they can be composed into a discrete simulation of a network in Appendix B. Continuous-time variants of components are implicitly defined as being parameterised by the number of steps per unit time $g$.

#### Utility

We define the **bytestream history** of a sender $s$ at time $t$ to be the string of tuples $(t_{s,1}, t_{s',1})(t_{s,2}, t_{s',2})\ldots$ where

- $t_{s,i}$ is the time that byte $i$ was first offered to a sender $s$ by the traffic source
- $t_{s',i}$ is the time that byte $i$ was first offered to the traffic source by the opposite sender $s'$

- the string contains tuples representing every such byte offered to the traffic source by time $t$

We now define **Utility** as a the set of functions $\mathbb{R} \times \left((\mathbb{R} \times \mathbb{R})^*\right)^* \to \mathbb{R}$. A utility function takes a global time $t$ and the bytestream history of all senders at time $t$, and returns a value representing the utility at that time. Ideally the function should be weighted to give a lower weighting to events further in the past (e.g. using exponential discounting).

### CCA-Template

We define a **CCA-Template** as a tuple $(\mathcal{O}, \mathcal{A}, F, L)$ where

- $\mathcal{O}$ is the set of possible observations,
- $\mathcal{A}$ is the set of possible actions, and
- $F : \left(\mathcal{O} \overset{\text{pf}}{\to} \mathcal{A}\right) \to \text{CCA}_L$ is a function that takes a policy $\pi \in \left(\mathcal{O} \overset{\text{pf}}{\to} \mathcal{A}\right)$ and returns a $\text{CCA}_L \ni C$ that calls $\pi$ at most once each time $C$ is called.

### Continuous AD′-POMDP

We can construct a continuous AD′-POMDP from a tuple $(T, D, u, \alpha)$ where

- $T$ is a CCA-Template $(\mathcal{O}', \mathcal{A}', F, L)$,
- $D$ is a probability distribution over $\text{Network}_L$,
- $u$ : Utility is a utility function, and
- $\alpha \in \mathbb{R}$ is a discount rate.

Details of how this is done can be found in Appendix B.3, but a key idea is that we treat utility as a "rate of reward".

Finally, we can define CGCC.

**Definition 5**

We define **the problem of learning congestion control for $\boldsymbol{T}$** where $T$ is a CCA-Template $(\mathcal{O}, \mathcal{A}, F, L)$ as follows.

Given $(D, u, \alpha)$, find a policy $\pi \in \left(\mathcal{O} \overset{\text{pf}}{\to} \mathcal{A}\right)$ that maximises the expected discounted return of the continuous AD′-POMDP constructed from $(T, D, u, \alpha)$.

# 3 Implementation

Now that we've covered the project foundations and technical preliminaries, we can dive into the design and implementation of **FlowForge** – a framework for learning congestion control. In this chapter, we start by covering the core framework, explore how Remy is implemented using this framework, and investigate how reinforcement learning can be used as an alternative training method. Simplified snippets of Rust code are frequently included – we recommend reading "The Book" [Klabnik and Nichols, 2024] to fully understand these.[6]

## §3.1 FlowForge

In this section, we explore the design of the core framework, covering the structure of the repository, a typical workflow, and tradeoffs made to adapt our theoretical model of learning congestion control into a practical one.

### §3.1.1 Introduction

The project is divided into four key components:
- a **core library**, which presents an API for interfacing with the framework
- a **command line interface**, which allows users to interact with the framework
- a **dynamic library**, which provides `C` bindings for loading and running policies
- a **modified version of `ns-2`** (as mentioned in Section 2.1.1), used for evaluating the learned CCAs

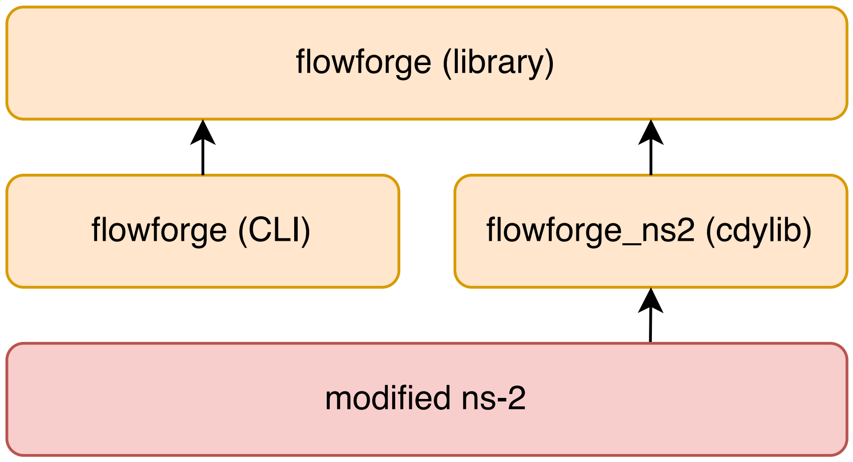


**Figure 1**: Dependency graph between the core framework, CLI, dynamic library, and `ns-2`.

[6] Particularly sections 4 (ownership), 10.2 (traits), 10.3 (lifetimes), and 19.2 (advanced traits). A free online copy is linked in the bibliography.

A typical workflow involves using the CLI to train a policy from some configuration files, evaluating the policy using `ns-2`, and then deploying the policy to production.

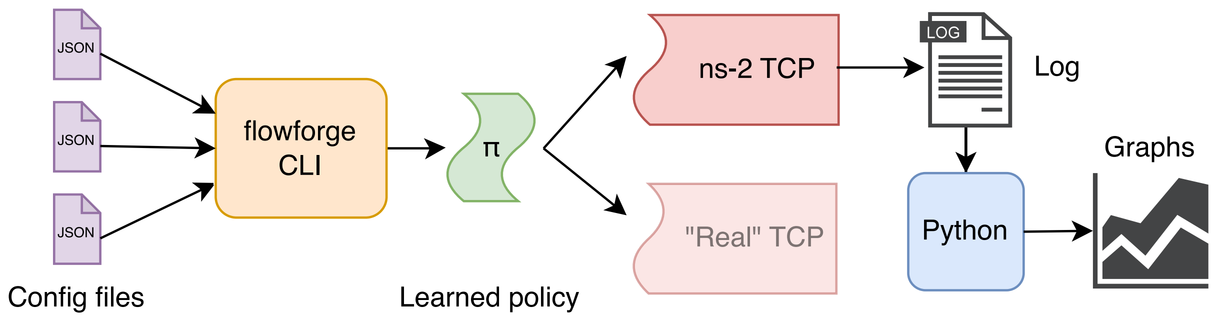


**Figure 2**: A typical workflow using the flowforge CLI. Note that this project does not aim to provide a “real” integration with TCP.

### §3.1.2 Abstractions

In order for the framework to be useful, we must provide abstractions that are flexible enough to support a wide range of approaches for learning congestion control, whilst being both practical and performant. We approach this by starting with our theoretical model for learning congestion control (Section 2.2.3) and then relaxing the details until we achieve something practical.

**Simulation.** In our formal model, a network simulation is an autonomous, probabilistic, infinite-state machine that steps $x$ times a second as $x \to \infty$. This presents a couple of issues.

1. **Infinite states** – since computers have finite memory, our practical model must have finite states.
2. **Stepping infinitely often** – computers cannot perform an infinite number of operations per second, so our practical model must make “jumps” in time to make progress.

Like many other programs in this field, we chose to use **discrete-event simulation (DES)**, which models the world as a sequence of instantaneous events, between which nothing happens. We discuss the implementation details of this in Section 3.2, but two key details are that each simulation has a *unique identifier* (denoted `'sim`) and a *message type* (denoted `G`, `G::Of<'sim>` or `E`).

**Quantities.** Our theoretical model has no concept of quantities – only real and natural numbers. To clarify intention and improve correctness, we perform dimensional analysis at compile time using the *newtype* idiom. This involves creating wrapper types for `Time`, `TimeSpan`, `Information`, and `InformationRate`, and then defining conversions into, between, and out of these types.

**Randomness.** In the theoretical model, we define components abstractly using independent probability distributions. There are two main issues with this.

1. **Directly representing complex distributions** (such as a probability distribution over state transitions) is impractical.
2. **True randomness** is both impractical and undesirable (as it prevents reproducibility).

The approach we took is to use a seeded pseudo-random number generator (PRNG), from which all randomness in the program is derived. First, we define an `Rng` struct using the fast `rand_xoshiro::Xoshiro256PlusPlus` PRNG, allowing the `Rng` to sample from simple distributions that implement the `rand_distr::Distribution` trait. We achieve reproducibility and practicality by deterministically seeding a single `Rng`, and then threading a mutable reference to it through the entire program.

Naively sharing an `Rng` across multiple threads using a `Mutex<Rng>` would introduce non-determinism in the form of OS scheduling behaviour. To work around this, we allow an `Rng` to spawn "child" `Rng`s, which can be safely sent to another thread. We also allow the spawning of "identical child factories", which can produce multiple identical `Rng`s for replayability purposes (e.g. tracking training progress by periodically evaluating the CCA using the same set of networks).

**Utility functions.** We originally represented utility functions as functions that take the entire bytestream history, and produce a single output. To make this more efficient, we first define `FlowProperties` to hold useful statistics about *packets* (not individual bytes) of a unidirectional flow. We also define the `FlowMeter` trait, to represent types that keep track of these statistics.

```
pub struct FlowProperties {
    pub throughput: InformationRate,
    pub rtt: Result<TimeSpan, NoPacketsAcked>,
}

pub trait FlowMeter {
    fn set_enabled(&mut self, time: Time);
    fn set_disabled(&mut self, time: Time);
    fn packet_received(&mut self, data: Information, rtt: TimeSpan, time: Time);
}
```

With these types, we now define `UtilityFunction` to operate on `FlowProperties`.

```
pub trait UtilityFunction: Sync {
    fn utility(&self, flows: &[FlowProperties]) -> Result<Float, NoActiveFlows>;
}
```

Whilst this significantly diverges from the formal model, every existing approach to learning congestion control that the author could find defines utility in terms of such statistics.

**CCA.** The formal model describes a CCA as a stateful function that is called once per round, takes arguments representing the sequence numbers of the packets that were sent or received in the previous round, and returns a `cwnd`. In fitting with the DES approach (whilst keeping close to the formal model), we represent a `Cca`[7] as a component that reacts to events

[7]This follows Rust naming convention for acronyms.

and can choose to be "ticked" at any point in the future. Upon each event, it returns a new `cwnd`.

```
pub trait Cca: Debug {
    fn initial_cwnd(&self, time: Time) -> u32;
    fn next_tick(&self, time: Time) -> Option<Time>;
    fn tick(&mut self, rng: &mut Rng) -> u32;
    fn packet_sent(&mut self, packet: PacketSent, rng: &mut Rng) -> u32;
    fn ack_received(&mut self, ack: AckReceived, rng: &mut Rng) -> u32;
}
```

**CCA-Template.** Our formal model describes a CCA-Template as a function that takes a policy $\pi \in \mathcal{O} \xrightarrow{\text{pf}} \mathcal{A}$ and returns a $\text{CCA}_L$. We relax the requirement that $\pi$ should be a function, allowing for clearer config-style policies.

```
pub trait CcaTemplate<'a>: Default {
    type Policy: 'a;
    type Cca: Cca + 'a;
    fn with(&self, policy: Self::Policy) -> impl Fn() -> Self::Cca + Sync;
}
```

**Network.** In the theoretical model, we describe a Network as being comprised of a Sender, an Interconnect-Behaviour, and a Traffic-Behaviour. Whilst our framework could enforce these components directly, to encourage maximum flexibility we decided to model a network as something that takes a `Cca` and a `FlowMeter` and builds a simulator.

```
pub trait Network<G>: Clone + Send where G: OfLifetime {
    fn populate_sim<'sim, 'a, C, F>(
        &self,
        builder: &SimulatorBuilder<'sim, 'a, G::Of<'sim>>,
        new_cca: impl Fn() -> C + Clone + 'a,
        rng: &'a mut Rng,
        new_flow_meter: impl FnMut() -> F,
    ) where C: Cca + 'a, F: FlowMeter + 'a, 'sim: 'a;
}
```

Our definition of learning congestion control includes a probability distribution over Network – we represent this as a `NetworkDistribution`, which implements `rand_distr::Distribution` over an associated `NetworkBuilder` type.

```
pub trait NetworkDistribution<G>: Distribution<Self::Network> + Sync
where G: OfLifetime {
    type Network: Network<G>;
}
```

**Trainer.** In Definition 5, we say that something *solves the problem of learning congestion control for a CCA-Template $T$* if it can take a probability distribution of networks and a utility function, and produce a policy for $T$ that maximises the expected discounted return. In FlowForge, we define a `Trainer` to represent such a "something".

```rust
pub trait Trainer {
    type Dna: Dna;
    type CcaTemplate<'a>: CcaTemplate<'a, Policy = &'a Self::Dna>;
    fn train<G: OfLifetime>(
        &self,
        network_config: &impl NetworkDistribution<G>,
        utility_function: &impl UtilityFunction,
        progress_handler: &mut impl ProgressHandler<Self::Dna>,
        rng: &mut Rng,
    ) -> Self::Dna;
}
```

Each Trainer has an associated Dna type, representing a policy that can be saved to and loaded from a file.

### §3.1.3 Repository overview

```
flowforge/
├── configs/                        // Default JSON configs
│   ├── eval/
│   ├── network/
│   ├── trainer/
│   └── utility/
├── examples/
│   └── (+3)
├── ns2/
│   ├── include/flowforge_ns2.h     // C header file for the dylib, used by ns-2
│   ├── src/lib.rs                  // Provides "extern" C bindings
│   ├── Cargo.toml
│   └── Cargo.lock
├── scripts/                       // For plotting progress and tracing simulations
│   └── (+3)
├── src/
│   ├── bin/flowforge/              // FlowForge CLI
│   │   ├── main.rs
│   │   ├── create_configs.rs
│   │   ├── evaluate.rs
│   │   ├── inspect.rs
│   │   ├── trace.rs
│   │   └── train.rs
│   ├── ccas/
│   │   ├── remy/
│   │   │   ├── mod.rs
│   │   │   ├── remy_dna.proto      // COPIED FROM THE ORIGINAL REMY REPO
│   │   │   ├── test_dna/           // FOR ROUND-TRIP TESTING, ALSO COPIED
│   │   │   └── (+5)
│   │   ├── remyr/
│   │   │   ├── mod.rs
│   │   │   ├── snapshots/          // For snapshot testing
│   │   │   └── (+2)
```

```
|   |   └── (+2)
|   ├── components/               // Reusable DES components
|   |   ├── senders/
|   |   |   ├── mod.rs
|   |   |   └── lossy.rs
|   |   └── (+6)
|   ├── networks/
|   |   ├── mod.rs
|   |   └── remy.rs
|   ├── quantities/               // For compile-time dimensional analysis
|   |   ├── information_rate.rs
|   |   └── (+4)
|   ├── trainers/
|   |   ├── snapshots/
|   |   ├── remy.rs
|   |   ├── remyr.rs
|   |   └── (+3)
|   ├── util/
|   |   └── (+6)
|   ├── eval.rs
|   ├── flow.rs
|   ├── lib.rs                    // Core abstractions
|   └── simulation.rs             // Discrete-event simulator
├── build.rs                      // Integrates protobuf-codegen with cargo build
└── (+ 5)
```

## §3.2 Discrete-event simulation

A core requirement of FlowForge is to be able to support a wide range of network behaviours through the use of a high-performance, flexible discrete-event simulator. We start by identifying some key features that the simulator should offer:

1. **Modularity** – simulations should be constructable from multiple components of different types, allowing reusability
2. **Communication** – components should be able to send messages to each other
3. **Events** – components should be able to schedule future events
4. **Flexible ownership** – the simulator should allow components with different ownership structures (e.g. owned, exclusively borrowed, or shared)
5. **Determinism** – the simulator should not introduce any non-determinism[8]

To take full advantage of Rust's type system, we would also like the simulator to enforce the following useful properties at compile time:

1. **Well-typed communication** – messages sent by a component must be able to be accepted by the recipient component
2. **Simulator isolation** – components in one simulation should not be able to send messages to components in another

[8]Other than that which is inherent to the component implementations.

The author couldn't find any existing Rust libraries that would have satisfied these requirements, and therefore made the decision to create such a library from scratch.

## §3.2.1 Ownership issues

In traditional programming languages, we could implement such a simulator using a list of heap-allocated objects, each with references to other objects that they can call methods on. This approach isn't possible in FlowForge due to Rust's strict ownership rules. There are two main issues:

1. **Reference cycles.** The recommended way to represent shared references is to use `Rc`, which causes memory leaks if there are cycles in the reference structure.
2. **Re-entrancy.** In safe Rust, we can never hold two mutable references to the same object at any one time. This causes issues with the traditional approach, as we would need to disallow the same object appearing twice in the function call stack – we couldn't have bi-directional communication between two components within the same timestep.

We address the first problem through the use of integer indices instead of references to refer to components.[9] The second issue is solved by moving the responsibility of message delivery from components to the event loop by using a message queue.

**Algorithm 1:** Core event loop

```
components ← $\langle c_0, \ldots, c_{N-1} \rangle$
next_tick ← $\text{null}^N$
for i $\in \mathbb{N}_N$ do
    next_tick[i] ← components[i].next_tick(0) # may be null
end for
while $\exists i \in \mathbb{N}_N.\ i = \arg\min_j(\text{next\_tick}[j])$ do
    (current, time) ← (components[i], next_tick[i])
    messages ← Queue(current.tick())
    next_tick[i] ← current.next_tick(time)
    while messages not empty do
        (dest, msg) ← messages.pop()
        current ← components[dest]
        messages.push_all(current.receive(msg, time))
        next_tick[dest] ← current.next_tick(time)
    end while
end while
```

[9]Surprisingly, using integer indices is considered an "idiomatic" approach and is used by popular libraries.

### §3.2.2 Type safety

Our goal is to ensure that components can only communicate with other components in the same simulation, and that any such communication is via type-checked interfaces. We start by defining a `ComponentId` to represent an index into the components list.

```
pub struct ComponentId<'sim> {
    index: usize,
    sim_id: Id<'sim>,
}
```

As we'll see later, we use the `generativity` crate to give each simulator instance a unique lifetime, denoted `'sim`. `Id` has an *invariant lifetime*,[10] which prevents `ComponentId`s from one simulation from being confused with `ComponentId`s from another *at compile time*.

We now define a `Message` type, which represents a generic message to be delivered to some component.

```
pub struct Message<'sim, E> {
    destination: ComponentId<'sim>,
    effect: E,
}
```

All messages handled by the simulator must have the same generic parameter `E`, to allow them to be stored in a message queue. However, we still want to restrict what types of messages individual components can receive. To reflect this, we define a `Component` trait. We automatically implement this trait for `&mut C` and `Rc<RefCell<C>>` where `C: Component<'sim, E>`, allowing flexible ownership structures.

```
pub trait Component<'sim, E> {
    type Receive;
    fn next_tick(&self, time: Time) -> Option<Time>;
    fn tick(&mut self, context: EffectContext) -> Vec<Message<'sim, E>>;
    fn receive(&mut self, e: Self::Receive, context: EffectContext)
        -> Vec<Message<'sim, E>>;
}
```

In this model, components accept messages of type `Self::Receive`, but must return messages of type `Message<'sim, E>`. To make this usable, we need to implement two methods of casting – one from specific message types to `Message<E>` (when sending a message), and another from `E` to the relevant `Component::Receive` (when receiving a message).

We implement the first type of casting through the concept of an `Address`.

```
pub struct Address<'sim, I, E> {
    create_message: Rc<dyn Fn(I) -> Message<'sim, E> + 'sim>,
}

impl<'sim, I, E> Address<'sim, I, E> {
    fn new(component_id: ComponentId<'sim>) -> Address<'sim, I, E>
```

[10]The type is just a wrapper over `PhantomData<fn(&'id ()) -> &'id ()>`, which is invariant over `'id`.

```
    where E: From<I>;

    pub fn cast<J>(self) -> Address<'sim, J, E>
    where J: 'sim, I: From<J> + 'sim, E: 'sim;

    pub fn create_message(&self, effect: I) -> Message<'sim, E>;
}
```

The second type of casting (from `E` to the relevant `Component::Receive`) is performed by an `AssertWrapper`, which wraps a `Component<E, Receive=R> where E: HasVariant<R>` to provide a `Component<E, Receive=E>` interface.[11]

```
struct AssertWrapper<C>(C);

impl<'sim, C, E> Component<'sim, E> for AssertWrapper<C>
where C: Component<'sim, E>, E: HasVariant<C::Receive> {
    type Receive = E;
    fn next_tick(&self, time: Time) -> Option<Time> { ... }
    fn tick(&mut self, time: Time) -> Vec<Message<'sim, E>> { ... }
    fn receive(&mut self, e: E, time: Time) -> Vec<Message<'sim, E>> {
        let e = e.try_into().unwrap(); // should never panic
        self.0.receive(e, time)
    }
}
```

By ensuring that only the `simulation` module can construct `Address`es or `Message`s, we can guarantee at compile time that messages returned to the event loop can be accepted by the recipient.[12]

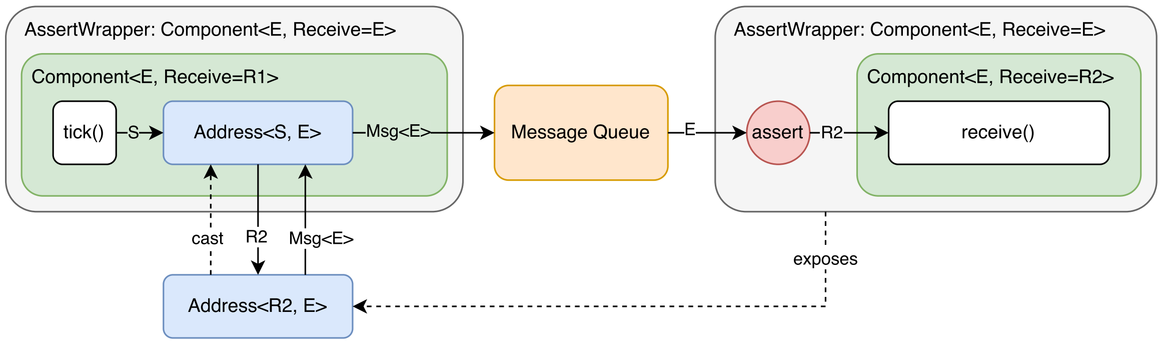


**Figure 3**: An overview of the mechanism used to ensure type-safe communication between components. In this example, `E` is the global message type, `R2` $\sqsubseteq$ `E` is the type that the component on the right accepts, and `S` $\sqsubseteq$ `R2` is a type that the first component can produce. Our design ensures at compile time that the `assert` conversion from `E` to `R2` doesn't fail.[12]

[11]`HasVariant<R>` is just a trait alias for `From<R> + TryInto<R>`.

[12]Assuming that all `Component::Receive` types are reasonably-implemented variants of `E` – i.e. for all receive types `R` $\sqsubseteq$ `E` we have `r.into::<E>().try_into::<R>().unwrap() == r`.

We then define our `Simulator` as an `Id`, a `TickQueue`, and a list of `AssertWrappers`. Similar to the pseudo-code, our `TickQueue` implementation uses a `vec_map::VecMap` to store next-tick times, as profiling showed that common priority queue implementations were inefficient due to the internal use of `HashMaps` (even when using the fast `rustc_hash::FxHash` hasher). The message queue uses a `std::collections::VecDeque`.

```
pub struct Simulator<'sim, 'a, E> {
    id: Id<'sim>,
    components: Vec<Box<dyn Component<'sim, E, Receive = E> + 'a>>,
    tick_queue: TickQueue,
    ...
}

impl<'sim, 'a, E> Simulator<'sim, 'a, E>
{
    pub fn tick(&mut self) -> bool; // returns false if finished
    pub fn time(&self) -> Time;
}
```

So far, we've made an effort to support cyclical messaging. However, this requires components to be constructed from each other's (or their own) address. To support this, we use the *builder pattern* in the form of a `SimulatorBuilder` struct. This allows as to insert components directly, or reserve slots (and the corresponding address) so that they can be inserted later.

```
pub struct SimulatorBuilder<'sim, 'a, E> {
    id: Id<'sim>,
    components: RefCell<Vec<Option<Box<dyn Component<'sim, E, Receive = E> + 'a>>>>,
}

impl<'sim, 'a, E> SimulatorBuilder<'sim, 'a, E> {
    pub fn new(guard: Guard<'sim>) -> SimulatorBuilder<'sim, 'a, E>;

    pub fn insert<C>(&self, component: C) -> Address<'sim, C::Receive, E>
    where
        C: Component<'sim, E> + 'a,
        E: HasVariant<C::Receive> + 'sim;

    pub fn reserve_slot<'b, C>(&'b self) -> ComponentSlot<'sim, 'a, 'b, C, E>
    where
        C: Component<'sim, E>,
        E: From<C::Receive> + 'sim;

    pub fn build(self) -> Result<Simulator<'sim, 'a, E>, EmptySlot>;
}
```

The `generativity::Guard` constructor argument for `SimulatorBuilder` is what enables us to enforce a unique lifetime per simulator, which allows the `Ids` to prevent simulators from mixing. In Appendix C, we provide an example of how we can use the module to construct a DES, and show how the compiler enforces simulator isolation and well-typed communication.

## §3.3 Oxidising Remy

In this section, we show how Remy fits into the abstractions offered by FlowForge (as in Section 3.1.2), and discuss the details of our practical implementation.

### §3.3.1 Utility

In the Remy paper, Winstein and Balakrishnan describe *alpha-fairness* as a method of representing an explicit fairness-efficiency tradeoff for users of a shared resource. They explain how some traditional methods described the total utility as $\sum_i U_\alpha(r_i)$, where $r_i$ is the amount of resource received by user $i$, and

$$U_\alpha(x) = \frac{x^{1-\alpha}}{1-\alpha}$$

As $\alpha \to 1$, in the limit $U_\alpha(x)$ becomes $\log x$. They develop this idea to give the utility of each flow as

$$U(\text{throughput}, \text{rtt}) = U_\alpha(\text{throughput}) - \delta \cdot U_\beta(\text{rtt})$$

where $\alpha$ and $\beta$ represent the fairness-efficiency tradeoffs in throughput and delay respectively, and $\delta$ expresses the relative importance of delay compared to throughput. **Proportional throughput-delay fairness** is a special case of this utility when $\alpha = 1$ and $\beta = 1$.

In FlowForge, we represent this class of utility functions with an `AlphaFairness` type.

```
pub struct AlphaFairness {
    alpha: Float,               /// Fairness of throughput
    beta: Float,                /// Fairness of round-trip delay
    delta: Float,               /// Relative importance of delay
    worst_case_rtt: TimeSpan,   /// Worst-case (max) round-trip delay
}

impl UtilityFunction for AlphaFairness { ... }
```

We use the `worst_case_rtt` field to ensure that the minimum utility for each flow is 0, by subtracting the utility when `bandwidth=0` and `rtt=worst_case_rtt`. The total utility is given as the average utility of each flow.

In Remy, only the final overall bandwidth and RTT are considered by the utility function – therefore, we define an `AverageFlowMeter: FlowMeter` that keeps track of `FlowProperties` averaged over the entire history.

### §3.3.2 CCA

In Remy, the algorithm learns a $\mathbb{R}^3 \to \mathbb{R}^3$ decision tree that maps from `Points` to `Actions` every time an ack is received. To improve extensibility, we define an abstract `RemyPolicy` trait.

```
pub struct Point {
    /// EWMA of time between received acks
    pub ack_ewma: TimeSpan,
    // EWMA of time between sender timestamps in the received acks
    pub send_ewma: TimeSpan,
    // Ratio of last received RTT to minimum received RTT
    pub rtt_ratio: Float,
}

pub struct Action {
    pub window_multiplier: Float,
    pub window_increment: i32,
    pub intersend_delay: TimeSpan,
}

pub trait RemyPolicy {
    fn action(&self, point: &Point) -> Option<Action>;
}
```

We then provide a decision tree implementation for this interface. Like the simulator, we use integer indices to refer to nodes in the tree – this time it allows us to retain mutable handles to nodes whilst the tree is being borrowed.

```
pub struct Cube { pub min: Point, pub max: Point }

pub enum RuleTreeNode {
    Node {
        domain: Cube,
        children: Vec<usize>,
    },
    Leaf {
        domain: Cube,
        action: Action,
        optimized: bool, // used in training
    },
}

pub struct RuleTree {
    root: usize,
    nodes: Vec<RuleTreeNode>,
}
```

To use `RemyPolicy`s in a network, we define a `RemyCca<T>` that is generic over `T: RemyPolicy`. The CCA keeps track of the current `Point`, and calls the policy each time an ack is received. The resulting action is then used to update internal `cwnd` and `intersend_delay` values. Since the generalised `Cca` interface only allows returning the `cwnd`, `RemyCca` emulates intersend delay by returning `cwnd=0` if it hasn't been at least `intersend_delay` since the last packet was sent. We also define a corresponding `RemyCcaTemplate<T>` that implements `CcaTemplate<Policy=T, Cca=RemyCca<T>>`.

### §3.3.3 Network

The TCP ex Machina paper focused on optimising CCAs for networks with a *dumbbell* topology, in which multiple senders communicate with multiple receivers on the opposite side of a link. In their custom simulator, the authors used three key approximations:

1. **Magnet topology.** Instead of simulating a dumbbell network directly, the authors chose a computationally-cheaper receiver-less approximation we call a *magnet* topology.

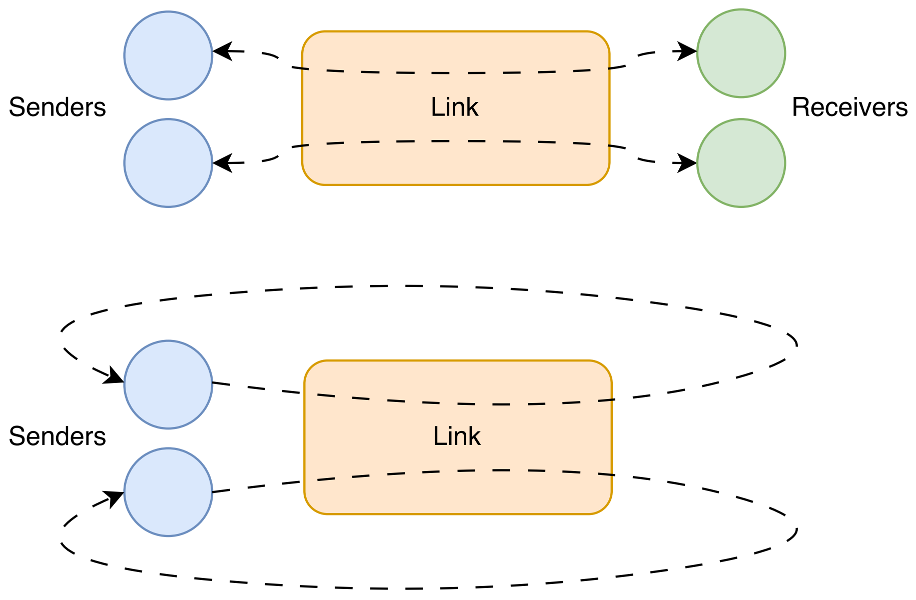


**Figure 4**: A comparison between a dumbbell network architecture (top) and the "magnet" approximation used by Remy (bottom).

2. **Binary flows**. Network traffic was approximated as a series of flows that alternated between "on" and "off" states, with the amount of time in each state being drawn from on-time and off-time distributions.
3. **Lossy senders**. Instead of supporting actual loss-recovery mechanisms, the Remy authors used a lossy sender during training (dropped packets were simply ignored).

In FlowForge, we apply all these approximations to our `RemyNetwork` implementation of `Network<G>`.

```
pub struct RemyNetwork {
    pub rtt: TimeSpan,
    pub packet_rate: InformationRate,
    pub loss_rate: Float,
    pub buffer_size: Option<Information>,
    pub num_senders: u32,
    pub off_time: PositiveContinuousDistribution<TimeSpan>,
    pub on_time: PositiveContinuousDistribution<TimeSpan>,
}
```

To implement `Network::populate_sim`, we first define `Toggler`, `LossySender`, and `Link` components, and then connect them in the following configuration.

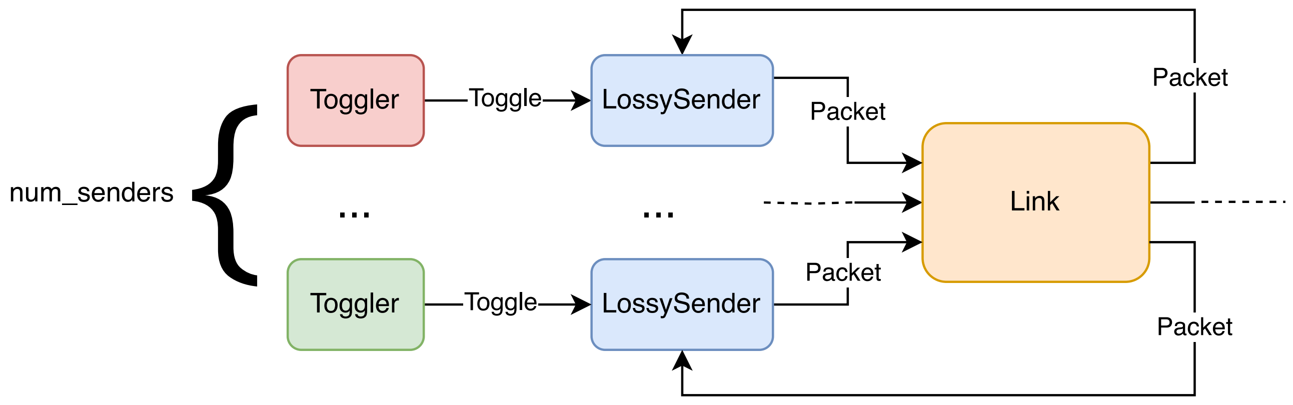


**Figure 5**: Components and message types of a `RemyNetwork`.

We also define a probability distribution of these networks `RemyNetworkDistribution: NetworkDistribution<G, Network=RemyNetwork>`. We don't attempt to represent "distributions of distributions" for `on_time` and `off_time` – these are passed directly to the sampled `Network`.

```
pub struct RemyNetworkDistribution {
    pub rtt: PositiveContinuousDistribution<TimeSpan>,
    pub bandwidth: PositiveContinuousDistribution<InformationRate>,
    pub loss_rate: ProbabilityDistribution,
    pub buffer_size: Option<DiscreteDistribution<Information>>,
    pub num_senders: DiscreteDistribution<u32>,
    pub off_time: PositiveContinuousDistribution<TimeSpan>,
    pub on_time: PositiveContinuousDistribution<TimeSpan>,
}
```

### §3.3.4 DNA

`RemyDna` is defined as a simple wrapper around a `RuleTree`. To maintain compatibility with the original Remy program, we need to be able to save and load `RemyDna` in the original protobuf format. We decided to use the `protobuf-codegen` crate to auto-generate Rust code corresponding to the original `remy_dna.proto` file, and then wrote translations between the two formats. The translation is validated using round-trip testing – we convert an existing folder of Remy DNA to our format and back again, and check that it remains unchanged. The codegen is cleanly integrated into the `cargo` build system using a `build.rs` file.

### §3.3.5 Trainer

Finally, we defined a `RemyTrainer` struct that implements `Trainer` with `Dna=RemyDna` and `CcaTemplate=RemyCcaTemplate<&RemyDna>>`.

**Algorithm 2:** Remy algorithm for learning congesting control

```
1  Initialise dna as a rule tree with a single rule.
2  for i=1 to rule_splits do
3      if i > 1 then
4          count(dna).most_used_rule().split()
5      end if
6      while ∃leaf. leaf=count(dna).most_used_unoptimised_rule() do
7          repeat
8              changed ← false
9              for new_action ∈ possible_improvements(leaf) do
10                 new_dna ← dna.with_replaced_action(leaf, new_action)
11                 new_score ← eval(new_dna, test_set)
12                 if new_score > old_score then
13                     (dna, old_score, changed) ← (new_dna, new_score, true)
14                 end if
15             end for
16         until not changed;
17         leaf.mark_optimised()
18     end while
19     dna.mark_all_unoptimised()
20 end for
```

As part of the `RemyTrainer::train` implementation, we define two `RemyPolicy` wrappers around `&RuleTree`, which can be seamlessly used with `RemyCca<T>`:

1. **`CountingRuleTree`** to record the number of times that each rule is used (Line 4, Line 6)
2. **`AugmentedRuleTree`** to override a particular rule with a provided action (Line 10)

Potential improvements are evaluated in parallel using the `rayon` library.

When implementing the algorithm, we noticed an inefficiency. Due to the total `Point` space being much larger than the policy inputs used in practice, the algorithm spent a large amount of time optimising and re-optimising the only used rule, until the point where more than one rule becomes used. To address this, we add an optional **drill-down optimisation** that only optimises the top-level rule once, and then repeatedly splits the most used rule until more than one rule is used.

## §3.4 Reinforcing Remy

In this section, we discuss how we implement **Remyr** – a novel trainer which uses RL to train Remy-inspired CCAs. As alluded to in Section 2.2.1, our strategy is to represent the policy as a MLP, which is then optimised using MAPPO.

### §3.4.1 Differentiable policies

We start by representing a deterministic policy as an MLP $\boldsymbol{\mu} \in \text{MLP}[3,3]$. `RemyPolicy` is implemented by translating from `Point` to $[-1,1]^3$, applying $\boldsymbol{\mu}$, and then translating back from $[-1,1]^3$ to `Action`. We also represent a stochastic policy $\boldsymbol{\pi}$ as a function of a deterministic policy $\boldsymbol{\mu}$, a diagonal multivariate Gaussian $\mathcal{N}(0, I_3)$, and a vector of standard devations $\boldsymbol{\sigma} \in \mathbb{R}^3$.

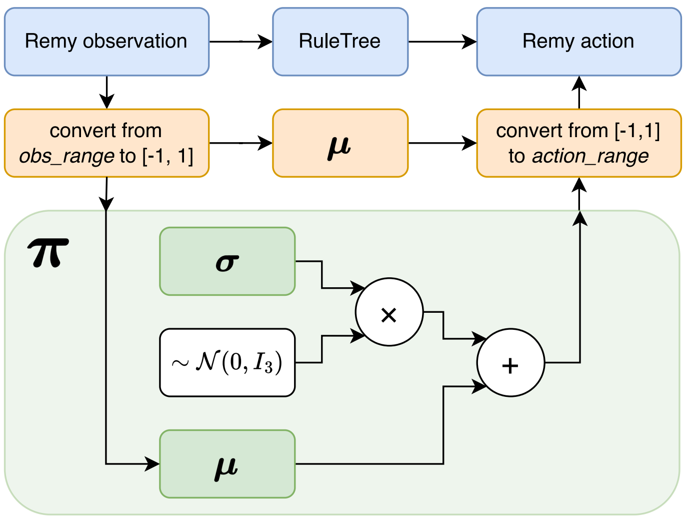


**Figure 6**: Diagram showing three ways to convert between Remy observations and actions – using an original Remy rule tree, using a deterministic Remyr policy $\boldsymbol{\mu}$, and using a stochastic policy $\boldsymbol{\pi} = (\boldsymbol{\mu}, \boldsymbol{\sigma})$.

Using the `dfdx` library [Lowman, 2022], we can represent $\boldsymbol{\mu}$ as follows.

```
pub const OBSERVATION: usize = 3;
pub const ACTION: usize = 3;

pub type PolicyArchitecture = (
    (LinearConfig<Const<OBSERVATION>, usize>, Tanh),
    (LinearConfig<usize, usize>, Tanh),
    (LinearConfig<usize, Const<ACTION>>, Tanh),
);
```

This corresponds to a MLP with an input layer of size `OBSERVATION`, two hidden layers (the sizes of which are determined at runtime), and an output layer of size `ACTION`. `Tanh` is used as the non-linear activation function.

We then define `RemyrDna` to represent a deterministic policy. Observations/actions are converted to/from $[-1, 1]^3$ by clamping the observations/actions between `min` and `max` values, and linearly mapping $[-1, 1]$ to $[\min, \max]$ for each dimension.

```
pub struct RemyrDna {
    pub min_point: Point,
    pub max_point: Point,
    pub min_action: Action,
    pub max_action: Action,
    pub policy: PolicyNetwork<Cpu>,
}
```

This `RemyPolicy` type is used during evaluation, to maximally exploit the learned policy.

During training, however, we use a `RolloutWrapper: RemyPolicy`, which wraps `dna: &RemyrDna` and `stddevs: &Tensor1D<ACTION>` to represent a stochastic policy (as in Figure 6). Due to the inherently delayed rewards and high-frequency sampling of the policy, the law of large numbers reduces the variation in exhibited behaviour, making it difficult to determine "bad" actions from "good" ones. To increase the impact of individual samples, during training we repeat each action $n$ times, where $n$ is drawn from a random distribution each time.

### §3.4.2 PPO

PPO uses gradient ascent to maximise the surrogate objective

$$L_t^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(\text{ratio}_t(\theta)\hat{A}_t,\ \text{clip}(\text{ratio}_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t\right)\right]$$

where $\hat{A}_t$ is an estimate for the advantage at time $t$, $\varepsilon$ is a hyperparameter, and

$$\text{ratio}_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

In Remyr, we estimate the advantage as

$$\hat{A}_t = R_t - V^\pi(s_t)$$

where $R_t$ is the *reward-to-go* at time $t$ (see Section 3.4.4 for more detail). This requires an approximation of the value function, and so we use a separate value function $V_\theta(s)$ in an *actor-critic* style. The value function is trained to minimise surrogate loss

$$L_t^{\text{VF}} = (V_\theta(s_t) - R_t)^2$$

To encourage exploration, PPO also attempts to maximise an entropy bonus. Since we describe our stochastic policy using a diagonal multivariate Gaussian, the entropy of a given sample is given by

$$S(s_t) = \sum_{i=1}^{\text{ACTION}} \frac{1}{2} \ln 2\pi e \boldsymbol{\sigma}_i^2$$

where $\pi$ is the constant (not a policy).

The two surrogate objectives are combined with the entropy bonus to form an overall objective

$$L_t(\theta) = \hat{\mathbb{E}}_t \left[ L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S(s_t) \right]$$

for coefficients $c_1$ and $c_2$.

### §3.4.3 MAPPO

The defining feature of MAPPO is that it gives the critic access to global state. In Remyr, we use a technique similar to the *agent-specific global state* mode proposed in the original paper, by feeding the value function a concatenation of local observations and some key global state (in our case, a single value representing 1/active_senders). Note that the output of the value function is unconstrained, due to the lack of activation function in the last layer.

```
pub const GLOBAL_STATE: usize = 1;
pub const AGENT_SPECIFIC_GLOBAL_STATE: usize = OBSERVATION + GLOBAL_STATE;

pub type CriticArchitecture = (
    (LinearConfig<Const<AGENT_SPECIFIC_GLOBAL_STATE>, usize>, FastGeLU),
    (LinearConfig<usize, usize>, FastGeLU),
    (LinearConfig<usize, Const<1>>,),
);
```

### §3.4.4 Discretised continuous discounting

Unlike many RL approaches to congestion control, Remyr is *reactive* – the policy is called at arbitrary times in response to received packets. This allows the policy to indirectly control the frequency at which it is called. We found that this causes simple exponential discounting methods to provide *perverse incentives*, harming stability and convergence.

To improve stability, we formulate a discounting method without these perverse incentives. We start by augmenting Equation 2.4 to give a definition for **continuous reward-to-go**.

$$R_t = \int_t^T e^{-\alpha(\tau - t)} u_\tau \, \mathrm{d}\tau$$

Next, consider an action at $t$, and let $t'$ be the time of the next action (if it exists, else $T$). Splitting the above integral gives us

$$R_t = \int_t^{t'} e^{-\alpha(\tau - t)} u_\tau \, \mathrm{d}\tau + \int_{t'}^T e^{-\alpha(\tau - t)} u_\tau \, \mathrm{d}\tau$$

where the second part can be written as

$$\int_{t'}^{T} e^{-\alpha(\tau-t)} u_\tau \,\mathrm{d}\tau = e^{-\alpha(t'-t)} \int_{t'}^{T} e^{-\alpha(\tau-t')} u_\tau \,\mathrm{d}\tau = e^{-\alpha(t'-t)} R_{t'}$$

If we make the simplifying assumption that $\forall \tau \in [t, t').\ u_\tau = u_{t'}$, then we get the recursive definition

$$R_t = \frac{1-\gamma}{\alpha} u_{t'} + \gamma R_{t'}$$

where $\gamma = e^{-\alpha(t'-t)}$. This turns out to be a discounting technique commonly used to learn semi-MDPs [Bradtke and Duff, 1994], which have a similar concept of *variable action duration* to our AD'-POMDP model of congestion control (Section 2.2.2).

**Algorithm 3:** MAPPO variant used by Remyr (see Appendix D)

```
for iteration=1, 2, … do
    for environment=1, …, N do                                  ▷ also known as a "rollout"
        Run policy $\boldsymbol{\pi}_{\theta_{\text{old}}}$ in environment for time $T$, taking discrete $r_t = u_t$
        Use discretised continuous discounting to calculate reward-to-go $R_t$
        Estimate advantage as $\hat{A}_t = R_t - \boldsymbol{v}(s_t, o_t)$
    end for
    Aggregate trajectories into a list of $\left(s_t, o_t, a_t, R_t, \hat{A}_t\right)$ tuples
    for epoch=1,…,num_epochs do
        Shuffle list and divide into num_minibatches
        for minibatch in minibatches do
            Use minibatch to optimise surrogate $L$ wrt $\theta$
            $\theta_{\text{old}} \leftarrow \theta$
        end for
    end for
end for
```

## §3.4.5 Implementation details

Due to the currently underdeveloped state of the Rust RL ecosystem, we implement PPO ourselves. We start by initialising an `AutoDevice` and constructing $\boldsymbol{\theta} = (\boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{v})$.

```
let dev = AutoDevice::default();
let mut theta = dev.build_module((
    self.hidden_layers.policy_arch(),
    Bias1DConfig(Const::<ACTION>),
    self.hidden_layers.critic_arch(),
));
```

GPU acceleration is supported through an optional `cuda` feature flag to `Cargo.toml`. If the feature is enabled, then `AutoDevice=Cuda` is used for optimisation. `Cpu` is always used during simulation due to the latency penalty of communicating with the GPU, but to mitigate this the *rollout* (trajectory collection) is performed in parallel using `rayon`.

In `dfdx`, many of the features normally hidden by `PyTorch` are exposed to the programmer. To perform auto-differentiation, `dfdx` has the concept of a `Tape`, which is responsible for recording operations. To accurately calculate gradients, a tape must be threaded through each differentiable operation by attaching it to one of the input tensors.

```
let batch_means = theta
    .0 // deterministic policy
    .forward(batch_observations.put_tape(OwnedTape::default()));
// Tape is now attached to batch_means
```

Sometimes manual tape manipulation is needed to re-use a tensor. In the following example, we use `batch_ratios` twice by splitting it into two copies – one with the original tape, and the other with an empty tape. We can perform operations on the two copies, and then implicitly merge the tapes in the `minimum` call.

```
let policy_loss = (-minimum(
    batch_ratios.with_empty_tape() * batch_advantages.clone(),
    clamp(batch_ratios, 1. - clip, 1. + clip) * batch_advantages.clone(),
))
.sum();
```

For each iteration, we calculate the total loss $L$, use the corresponding tape to perform back-propagation, and then use an optimiser to update $\boldsymbol{\theta}$ according to the computed gradients.[13]

```
let loss = policy_loss + critic_loss * self.value_function_coefficient
                       - entropy * self.entropy_coefficient;

let gradients = loss.backward(); // consumes the tape
optimizer.update(&mut theta, &gradients).unwrap();
```

To improve performance, we adopt several techniques from the *Stable Baselines 3* `ppo2` implementation [Raffin et al., 2021], including using an *Adam* optimiser [Kingma and Ba, 2014], performing *linear annealing* on both the Adam learning rate and the clip range $\varepsilon$, and normalising advantages at the minibatch level.

The author believes that this implementation is the first working example of continuous-action PPO in Rust.

[13]For a more complete picture of dataflow, see Appendix D.

# 4 Evaluation

Now that we've implemented the framework and accompanying algorithms, we need to fulfill the evaluation criteria that we specified in Section 2.1.2. Specifically, we need to evalute the *performance* of the learned CCAs, *compare* the Remy and Remyr approaches, and qualitatively evaluate the *utility* of the framework.

## §4.1 Performance of learned CCAs

Recall back to the requirements analysis, where we state that one of our core criteria is to evaluate the performance of Remy CCAs learned using the framwork against both

- CCAs trained using the original Remy implementation, and
- other traditional congestion control algorithms.

To fulfill the extension criteria, we extend this to also include Remyr CCAs.

### §4.1.1 Methodology

As in the TCP ex Machina paper, we use the framework to train some general-purpose CCAs. To give a fair comparison with the original Remy implementation, we use the same utility function and network distribution, and train for a similar number of iterations.

**Utility function.** For the utility function, we use proportional throughput-delay fairness (Section 3.3.1), and train four CCAs with $\delta = 0.1, 1, 10$ and 100. See Appendix E.2 for the exact configuration file used for $\delta = 1$.

**Network.** For the network, we use a `RemyNetwork` (Section 3.3.3) with the parameters given below. See Appendix E.1 for the exact configuration file used.

| Parameter | Value |
|---|---|
| Topology | `RemyNetwork` |
| RTT | Uniform from 100ms to 200ms |
| Bandwidth | Uniform from 10Mb/s to 20Mb/s |
| Loss rate | 0 |
| Buffer size | $\infty$ |
| Num senders | Uniform from 1 to 16 |
| Off time | Exponential with mean 5s |
| On time | Exponential with mean 5s |

**Table 1**: Description of the `NetworkDistribution` used during training.

**Remy trainer.** The parameters for the Remy trainer were mostly derived from the source code of the original Remy program, to allow a better comparison between implementations. The exact config file can be found in Appendix E.3.

**Remyr trainer.** The parameters for the Remyr trainer were inspired by the defaults used for the *Stable Baselines 3* `ppo2` implementation and refined through reasoned trial-and-error.

**Network simulator.** We evaluate the learned CCAs by modifying the `ns2` codebase used in the original TCP ex Machina paper. We consider three networks – one "ideal" network (where bandwidth and RTT are set to the mean of their respective training assumptions), and two networks with RTT and bandwidth at the limit of their assumptions.

| **Parameter** | **Value** |
|---|---|
| Topology | Dumbbell (not "magnet") |
| RTT | 100ms \| 150ms \| 200ms |
| Bandwidth | 20Mb/s \| 15Mb/s \| 10Mb/s |
| Loss rate | 0 |
| Buffer size | 1000 packets |
| Num senders | 8 |
| Off time | Exponential with mean 0.5s |
| On length | Exponential with mean 100kb |
| Simulation time | 100s |
| Iterations | 128 |

**Table 2**: Description of the `ns2` networks used during evaluation.

We compare the performance of the learned CCAs against the original Remy implementation ("OldRemy"), end-to-end schemes (NewReno, Compound, Cubic), and router-assisted schemes (XCP, sfqCoDel). The results are shown in Figure 7.

### §4.1.2 Analysis of results

Firstly, we can see that all three methods of learning control seem to perform better than the traditional CCAs – they have higher bandwidth, lower RTT, and generally lower variance.

Secondly, we see that the interpretation of $\delta$ seems to vary between FlowForge and OldRemy. We hypothesise that this is due to the original Remy implementation "normalising" the measured bandwidth by multiplying it by the number of active senders.

Thirdly, we see that our Remy implementation learns CCAs that perform similarly to the original. The exception is the 10Mb/s+200ms case, in which our implementation seems to favour higher bandwidths over lower queueing delays. This could be explained by the utility function being *proportional* - due to the higher base RTT, a fixed increase in queueing delay has a smaller impact on RTT (and hence utility). Overall, we believe our implementation satisfies the requirement that it produce CCAs which perform similarly to the original one.

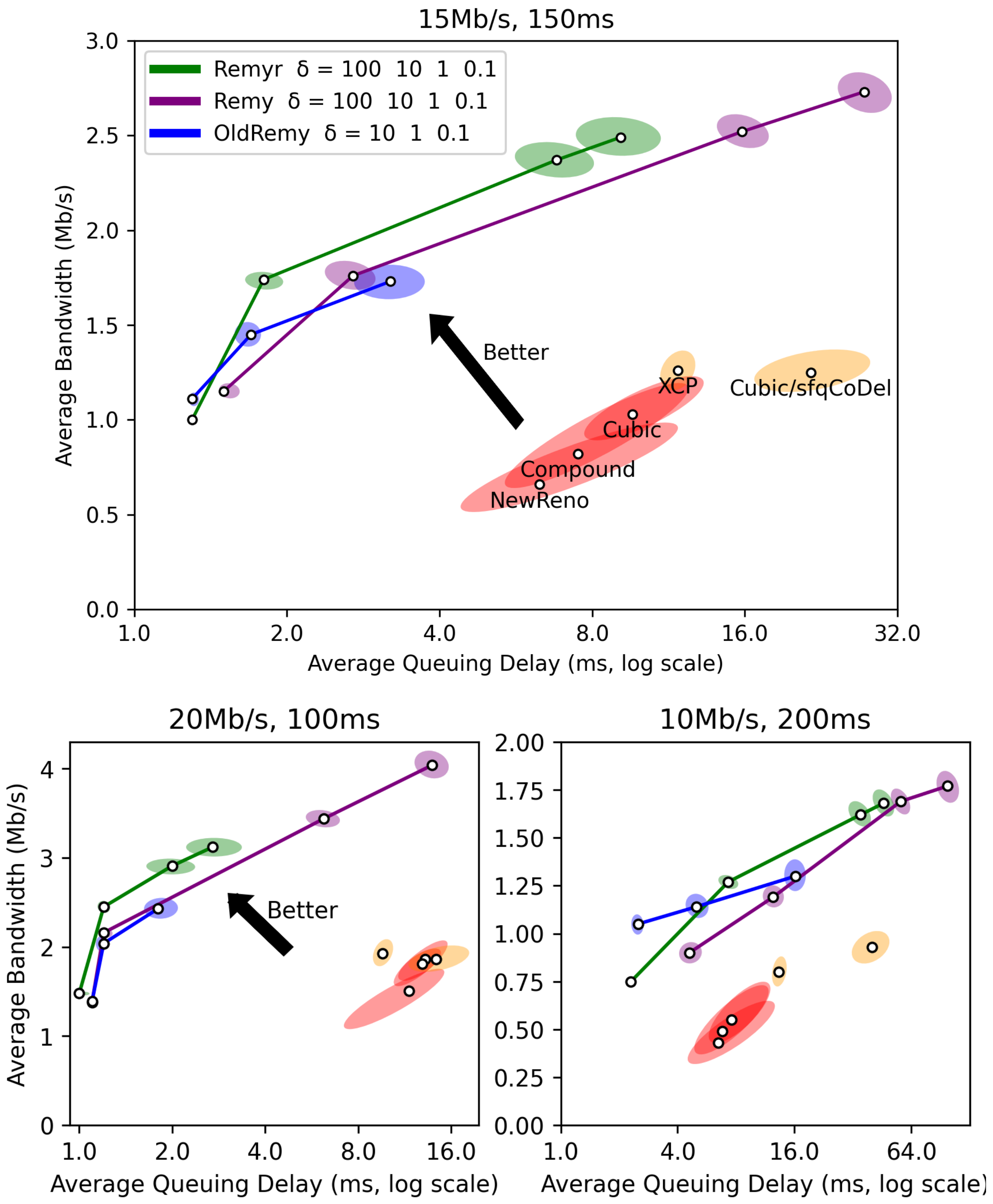


**Figure 7**: Medians and 1$\sigma$ covariance ellipses of various CCAs tested using `ns-2`. Each graph shows results for a different network. Configurations are described by Table 2, with the link bandwidth/RTT shown in the title.

Finally, we can see that the Remyr CCAs seem to perform better than our Remy CCAs across all networks. Although they do not reach as high a total bandwidth, the $\delta$-curve seems to be higher and further left, with the $\delta = 10$ case being strictly better in every network. We attempt to provide some reasoning for this behaviour in Section 4.2.

## §4.2 Comparison between approaches

When comparing the Remy and Remyr training processes, we notice two interesting details.

1. Remyr is around 100× faster to converge than Remy. For the CCAs used in Section 4.1, training takes around 20 hours for Remy, as opposed to 25 minutes for Remyr. We also found that Remyr is faster *even without GPU acceleration*. We hypothesise that this is due to Remy's over-optimisation of rules with lower usage – all rules are optimised for the same length of time in a given round, regardless of how frequently they are used.
2. Remy achieves higher evaluation scores in the FlowForge simulator, but not in `ns2`. We hypothesise that Remy selecting the best action out of thousands each time leads to the CCA overfitting the specific set of networks used for evaluating the potential improvements. This could also explain the large variance in score, even after the median has stabilised.

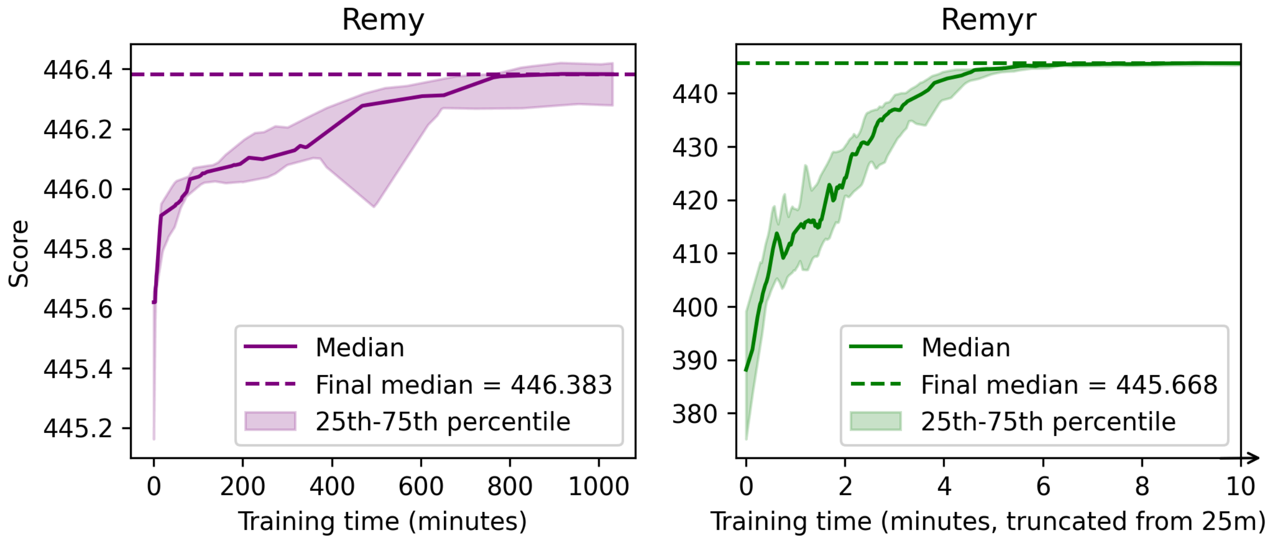


**Figure 8**: A comparison between the $\delta = 10$ Remy and Remyr training processes used in Section 4.1. The score is calculated as as the average utility of 30 simulation runs using the built-in FlowForge simulator. Note the differences in $x$ and $y$ scales.

The overfitting hypothesis is further supported by inspecting the policies directly. Figure 9 provides a visual comparison of a Remy policy and a Remyr policy. In particular, we notice that Remyr tends to learn smoother, simpler policies, whilst Remy tends to learn noisier ones. Despite their differences, the learned policies share some common features – this can be observed in the "increment" plot.

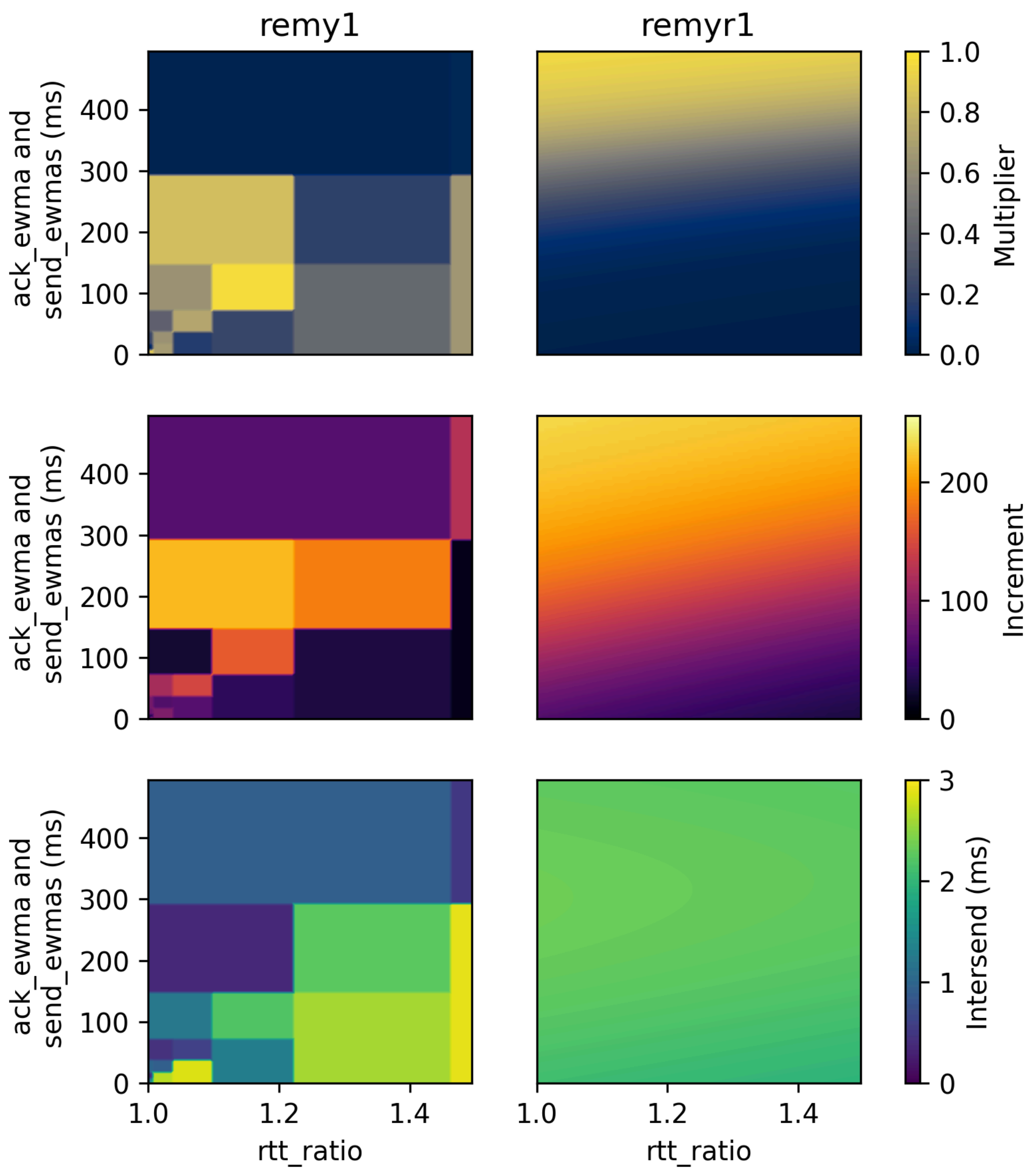


**Figure 9**: A visual comparison between the learned policies for $\delta = 1$. To condense the 3D domain into 2D, we only show points with `ack_ewma=send_ewma`.

Our evaluation so far has portrayed Remyr as better than Remy in every way – it is faster to train, and produces CCAs that are both more understandable and more performant. However, there is one significant downside to Remyr – policy sample time. As shown in Figure 10, querying the Remyr policy is **over 500× slower**. This constrains the theoretical maximum bandwidth (on an M1 Macbook Air) to somewhere in the region of 100Mb/s, with the practical maximum likely an order of magnitude or two lower.

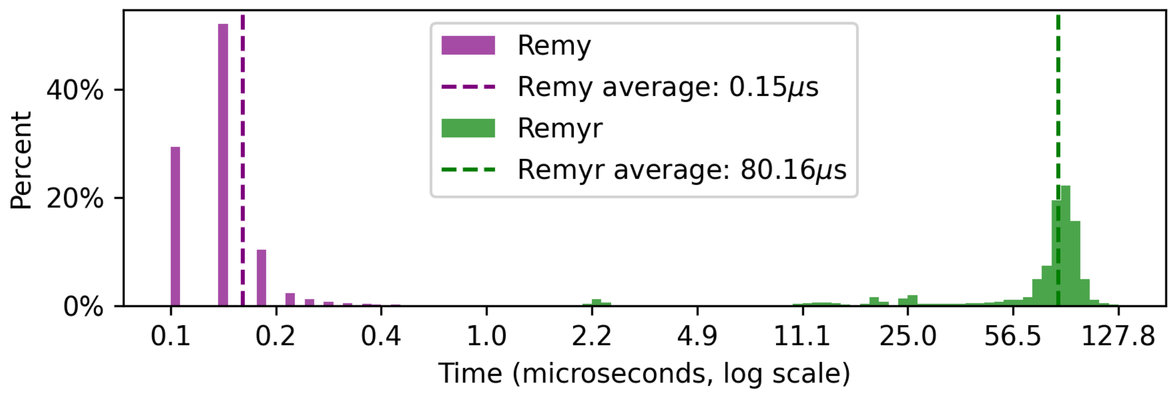


**Figure 10**: Distribution of execution times for Remy and Remyr policies on an M1 Macbook Air. Times were collected using the $\delta = 100$ policies used in Figure 7, which were run on a sample of 30 networks for 30 seconds each.

All is not lost, however. In their paper on RL for datacenter congestion control, Fuhrer et al. [2023] propose a method for distilling complex neural networks into lightweight decision trees. By doing this, they achieve a **500× inference time reduction**. We predict that this same algorithm could be applied to Remyr policies to convert them into Remy policies, retaining the original behaviour with much better practical performance.

## §4.3 Framework utility

In Section 2.1.2, we set out to qualitatively evaluate the utility of the framework for research and industry by considering

- *performance* of the core framework,
- *extensibility* given the provided abstractions,
- *reproducibility* of results, and
- *developer experience* when working with the framework.

### §4.3.1 Performance

During development, performance was a consideration at nearly every step. Examples include:

- choosing sensible abstractions when designing the practical framework
- *passing by reference* instead of moving where possible
- use of *monomorphisation* instead of dynamic dispatch
- use of *zero-cost abstractions*, such as the *newtype* idiom and *lifetimes as IDs*
- use of *profiling* to drive choice of algorithms and datastructures
- adding support for CPU and GPU parallelism (through `rayon` and `cuda` respectively)

However, optimisations were sometimes not taken if doing so would harm extensibility – for example, we implemented a generic DES rather than one specialised for Remy. Whilst we don't have a similar framework to quantitatively evaluate against, the author believes that any significant performance increases would likely require a reduction in extensibility.

### §4.3.2 Extensibility

In order for the framework to be extensible, we say that it should be both *flexible* (to support new ideas) and *reusable* (as otherwise there is little use for a framework).

**Flexibility.** We explicitly designed the theoretical framework to support an extremely wide range of end-to-end congestion control algorithms. Whilst conversion into a practical framework required restricting this somewhat, we believe that FlowForge still supports a very wide range of methods for learning congestion control. In addition to this, we support three flexible mechanisms of interfacing with the framework, which provide different control/simplicity tradeoffs: directly modifying the framework itself, using the framework as a library, and running built-in functionality via the CLI. We make particular effort to support the second method – for example, `Simulator` and `NetworkDistribution` are parameterised by message type, to avoid forcing a crate-defined default.

**Reusability.** Throughout the implementation, we decompose larger concepts into smaller, reusable ones. For example, simulator components from one `Network` can be reused by another, a new `UtilityFunction` can be used with an existing `Trainer`, a new `Trainer` can be used with an existing `NetworkDistribution`, and so on. Although we could have increased reusability by subdividing the `Network` trait into `Interconnect/Traffic/Sender` traits (as in the theoretical framework), this would have decreased flexibility.

Overall, we believe that we have achieved the goal of making the framework extensible, through the use of carefully chosen abstractions and re-usable components.

### §4.3.3 Reproducibility

As mentioned in Section 3.1.2, we make efforts to ensure that execution is deterministic (for a provided seed). This determinacy allows us to reliably reproduce Remy training and evaluation runs. In addition to determinacy, the CLI automatically includes all configuration files in the training progress output, encouraging researchers towards reproducible results.

Unfortunately, there are still situations where reproducibility is impacted, primarily where Remyr's use of `dfdx` is concerned. We observe differing (but deterministic) behaviour when switching between *arm64* and *x64* platforms, potentially due to architecture-specific optimisations. Furthermore, Remyr training using the `cuda` feature seem to be non-deterministic – this is likely a bug in `dfdx`.

Overall, we make an effort towards supporting reproducibility in the framework itself, but fall short in our Remyr implementation. Further investigation into `dfdx` is required before we can confidently claim to have fulfilled this criteria, but that would be beyond the scope of this project.

## §4.3.4 Developer experience

We believe that FlowForge provides an excellent developer experience when using and extending the framework.

**Developer tooling.** Setup is extremely easy, requiring just two commands – one to install Rust, followed by `cargo build`. Tests are run using `cargo test`, and the project uses default Rust tools (such as `rustfmt` and `clippy`) which are automatically supported by most IDEs.

**Type safety.** Throughout the project, we repeatedly apply the idea of making *invalid states unrepresentable* and *invalid operations impossible*. Examples include the type-safe communication of simulator components, compiler-enforced simulator isolation, and performing dimensionality analysis at compile time. All of these, combined with Rust's core rules, allow developers to modify and extend the framework with a high degree of confidence in correctness. Overall, the framework has a feeling of "if it compiles, it works", which is missing from existing Python and C++ approaches. However, this rigidity does come at the cost of some flexibility, impeding exploration and slowing the development of prototypes.

**Command line interface.** The provided CLI allows developers (and scripts) to interact with the framework without ever looking at the code. We use the `clap` argument parser to provide a polished CLI experience, with inputs taken in the form of JSON config files. These files are flexible and easy to understand, including support for units and prefixes (see Appendix E for examples).

## §4.3.5 Summary

Overall, we believe that the combination of high performance, fearless extensibility, and strong tooling allows the framework to provide excellent utility for software engineers looking to develop high-performance implementations of existing methods for learning congestion control. However, the strict typing and absence of support for popular ML libraries makes this framework less suitable for researchers who are looking to rapidly prototype new ideas.

# 5 Conclusions

This project was a success, fulfilling all the core criteria and the most ambitious of the proposed extensions.

## §5.1 Achievements

**Formalising CGCC.** In order to construct a practical framework, we first provided a *formalisation of computer-generated congestion control.* We did this by building a formal model for congestion control, and then framing it as a continuous AD'-POMDP – a new model which combines ideas from Dec-POMDPs with those from semi-MDPs.

**FlowForge.** Building on the formal model, we constructed a practical framework for learning end-to-end congestion control. The framework offers performance, fearless extensibility, and a good developer experience, allowing it to provide excellent utility for software engineers looking to develop high-performance implementations of methods for learning congestion control.

**Discrete-event simulator.** To emulate real-world network behaviours, we built a performant, single-threaded DES that offers strong compile-time guarantees. This allows us to create components which can be easily re-used across different network models.

**Remy.** We implemented the Remy algorithm for learning congestion control, adding a new *drill-down* optimisation to reduce training time, and used `ns2` to show that the produced CCAs perform similarly to those of the original implementation.

**Remyr.** We designed and implemented Remyr – a novel variant of Remy which uses RL. The technique used combined an algorithm designed for Dec-POMDPs (MAPPO) with a discounting mode designed for semi-MDPs. Not only did it *learn faster* than Remy, it also produced CCAs that *performed better* in simulation and were *easier to understand.* Pure Remyr CCAs are too compute-intensive to be practical in most cases – however, we believe that Remyr policies could be distilled into faster Remy ones which retain the same behaviour.

## §5.2 Future work

Aside from encouraging general use of the framework to explore methods of learning congestion control, we propose two specific ideas that could be applied to improve our Remyr algorithm.

1. **Learning to repeat.** In Remyr, we repeat actions for a random number of times to increase the variation in exhibited behaviour. However, we anticipate that this process could be improved by learning the number of times to repeat *on-policy*, as tried by Sharma et al. [2017].

2. **Distillation.** In the evaluation, we found that our Remyr policies were an order of magnitude slower to sample than their Remy counterparts. We hypothesised that *distilling the MLP* into a decision tree could provide a similar speedup to the one observed by Fuhrer et. al in 2023.

## §5.3 Reflections

Overall, the project has been both a challenging and satisfying experience. This was by far the most theoretically-intensive project I've worked on – it required venturing beyond my comfort zone, into areas that I had no prior experience in. In hindsight, developing a system to structure my research early on would have helped manage this more efficiently.

I think the most important lesson I learnt is conveyed by "Programming as Theory Building" [Naur, 1985]. Initially, I treated the project as the development of a program – code to do a task. However, only when attempting to provide a formalisation for Section 2.2.3 did I realise that I had unintentionally built a *poorly-reasoned theory* about CGCC. After explicitly developing this theory, I was able to identify and fix many naive decisions in the original codebase, resulting in a framework that is more extensible and easier to understand.

# Bibliography

Achiam, J., 2018. Spinning Up in Deep Reinforcement Learning.

Aslan, O., 2016. cargo-license [WWW Document].. URL https://github.com/onur/cargo-license

Boehm, B. W., 1988. A spiral model of software development and enhancement. Computer 21, 61–72.. https://doi.org/10.1109/2.59

Bradtke, S., Duff, M., 1994. Reinforcement Learning Methods for Continuous-Time Markov Decision Problems, in: Tesauro, G., Touretzky, D., Leen, T. (Eds.), Advances in Neural Information Processing Systems. MIT Press.

Ferrous Systems, 2019. flamegraph-rs [WWW Document].. URL https://github.com/flamegraph-rs/flamegraph

Fuhrer, B., Shpigelman, Y., Tessler, C., Mannor, S., Chechik, G., Zahavi, E., Dalal, G., 2023. Implementing Reinforcement Learning Datacenter Congestion Control in NVIDIA NICs, in: 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (Ccgrid). pp. 331–343.. https://doi.org/10.1109/CCGrid57682.2023.00039

Hemmati, M., Yassine, A., Shirmohammadi, S., 2015. A Dec-POMDP Model for Congestion Avoidance and Fair Allocation of Network Bandwidth in Rate-Adaptive Video Streaming, in: 2015 IEEE Symposium Series on Computational Intelligence. pp. 1182–1189.. https://doi.org/10.1109/SSCI.2015.170

Jay, N., Rotman, N., Godfrey, B., Schapira, M., Tamar, A., 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control, in: Chaudhuri, K., Salakhutdinov, R. (Eds.), Proceedings of the 36th International Conference on Machine Learning, Proceedings of Machine Learning Research. PMLR, pp. 3050–3059.

Jung, R., Jourdan, J.-H., Krebbers, R., Dreyer, D., 2017. RustBelt: securing the foundations of the Rust programming language. Proc. ACM Program. Lang. 2.. https://doi.org/10.1145/3158154

Kingma, D. P., Ba, J., 2014. Adam: A Method for Stochastic Optimization. CoRR.

Klabnik, S., Nichols, C., 2024. The Rust Programming Language [WWW Document].. URL https://doc.rust-lang.org/stable/book/

Liang, Y., Wu, H., Wang, H., 2022. ASM-PPO: Asynchronous and Scalable Multi-Agent PPO for Cooperative Charging, in: Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems, AAMAS '22. International Foundation for Autonomous Agents, Multiagent Systems, Virtual Event, New Zealand, p. 798–799.

Lowman, C., 2022. dfdx: shape checked deep learning in rust [WWW Document].. URL https://github.com/coreylowman/dfdx

Miller, B. L., 1968. Finite state continuous time Markov decision processes with an infinite planning horizon. Journal of Mathematical Analysis and Applications 22, 552–569.. https://doi.org/10.1016/0022-247X(68)90194-7

Nagle, J., 1995. Congestion control in IP/TCP internetworks. SIGCOMM Comput. Commun. Rev. 25, 61–62.. https://doi.org/10.1145/205447.205454

Naur, P., 1985. Programming as theory building. Microprocessing and Microprogramming 15, 253–261.. https://doi.org/https://doi.org/10.1016/0165-6074(85)90032-8

Oliehoek, F., Amato, C., 2016. A Concise Introduction to Decentralized POMDPs.. https://doi.org/10.1007/978-3-319-28929-8

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A., 2017. Automatic differentiation in PyTorch.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N., 2021. Stable-Baselines3: Reliable Reinforcement Learning Implementations. Journal of Machine Learning Research 22, 1–8.

Ronacher, A., 2019. insta [WWW Document].. URL https://insta.rs/

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017. Proximal Policy Optimization Algorithms. ArXiv.

Sharma, S., Lakshminarayanan, A. S., Ravindran, B., 2017. Learning to Repeat: Fine Grained Action Repetition for Deep Reinforcement Learning. ArXiv.

Shilliday, A., Taylor, J., Clark, M., Bringsjord, S., 2010. Provability-Based Semantic Interoperability for Information Sharing and Joint Reasoning, in: . pp. 109–128.. https://doi.org/10.3233/978-1-60750-581-5-109

Sivaraman, A., Winstein, K., Thaker, P., Balakrishnan, H., 2014. An experimental study of the learnability of congestion control. SIGCOMM Comput. Commun. Rev. 44, 479–480.. https://doi.org/10.1145/2740070.2626324

Tessler, C., Shpigelman, Y., Dalal, G., Mandelbaum, A., Haritan Kazakov, D., Fuhrer, B., Chechik, G., Mannor, S., 2022. Reinforcement Learning for Datacenter Congestion Control. SIGMETRICS Perform. Eval. Rev. 49, 43–44.. https://doi.org/10.1145/3512798.3512815

Winstein, K., Balakrishnan, H., 2013. TCP ex Machina: Computer-Generated Congestion Control, in: . pp. 123–134.. https://doi.org/10.1145/2534169.2486020

Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., Wu, Y., 2024. The surprising effectiveness of PPO in cooperative multi-agent games, in: Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS '22. Curran Associates Inc., New Orleans, LA, USA.

# A Models and algorithms

For convenience, we provide a summary of different environment models and corresponding algorithms that learn them. Our contribution is given in bold.

| Model | Stands for | Description | Example Algorithm |
|---|---|---|---|
| MDP | Markov Decision Process | Agent interacts with environment, has access to entire state. | TD(0) [14] |
| SMDP | Semi-MDP | MDP where actions have variable durations. | TD(0) with discretised continuous discounting |
| POMDP | Partially-Observable MDP | MDP where agent can only access a local observation. | PPO |
| Dec-POMDP | Decentralised POMDP | POMDP with multiple independent agents and a global reward. | Multi-agent PPO (MAPPO) |
| AD-POMDP | Asynchronous Decentralised POMDP | Dec-POMDP where each agent may or may not act each timestep, with per-agent rewards. | Asynchronous and Scalable Multi-agent PPO (ASM-PPO) |
| **AD'-POMDP** | **Asynchronous Decentralised POMDP** | **Dec-POMDP where each agent may or may not act each timestep.** | **MAPPO with discretised continuous discounting** |

[14]TD(0) only works for discrete state spaces.

# B Formalising CC

In this appendix, we sketch a formal model of congestion control, and then show how it can be converted into a continuous AD'-POMDP for use in our definition of learning congestion control (Section 2.2.3).

## §B.1 Stateful Functions

We define a **deterministic finite stateful function** $f : X \overset{\text{df}}{\to} Y$ to be a tuple $(M, F)$ where

- $M$ is a DFA with states $Q$ and input alphabet $X$, and
- $F : Q \to Y$.

To apply $f$ to $x \in X$, we first step $M$ using $x$ and then apply $F$ to the resulting state to get $f(x)$. This is essentially the same construction as a Moore machine.

We define a **probabilistic finite stateful function** $f : X \overset{\text{pf}}{\to} Y$ as a deterministic finite stateful function with the DFA replaced with a probabilistic finite automata (PFA).

We define a **deterministic infinite stateful function** $f : X \overset{\text{di}}{\to} Y$ similarly, by relaxing the constraint that $Q$ must be finite.

We define a **probabilistic infinite stateful function** $f : X \overset{\text{pi}}{\to} Y$ by combining the previous two augmentations.

Note that:

- all four classes of stateful function are closed under composition
- $\left(X \overset{\text{df}}{\to} Y\right) \subseteq \left(X \overset{\text{pf}}{\to} Y\right) \subseteq \left(X \overset{\text{pi}}{\to} Y\right)$
- $\left(X \overset{\text{df}}{\to} Y\right) \subseteq \left(X \overset{\text{di}}{\to} Y\right) \subseteq \left(X \overset{\text{pi}}{\to} Y\right)$

## §B.2 Simulation components

To make the model easier to understand, we break the simulation down into a series of modular components.

### §B.2.1 Interconnect behaviour

Consider a network with $N \in \mathbb{N}$ bidirectional TCP connections, composed of $2N$ senders.

We start by modelling all packets currently on the interconnect as a list of $2N$ buffers of size $B \in \mathbb{N}$, with the $i$th buffer representing the packets to be delivered to the $i$th sender. At each step, at most one packet can be pushed into each buffer. If we try to push a packet into a buffer that is already full, the packet is automatically dropped.

Now that we have a mechanism for packet entry to the interconnect, we need a mechanism to represent delay, duplication, loss, reordering, and delivery.

We define **Interconnect-Behaviour$_{N,B}$** to be the set

$$(\text{Packet} \cup \text{Null})^{2N} \overset{\text{pf}}{\to} (\mathbb{N}_{B+1} \times \mathcal{P}(\mathbb{N}_B) \times \mathcal{P}(\mathbb{N}_B))^{2N}$$

At each step, an interconnect behaviour receives a list of $(p_{\text{snd}})_s$ where

- $s$ is the index of a sender
- $p_{\text{snd}}$ is a packet that will be pushed to the buffer belonging to $s$ (if it isn't full)

It then returns a list of $(i, m, d)_s$ tuples where

- $i$ is the index of an item in the buffer belonging to $s$ to deliver (if the index is in range)
- $m$ is a set of indices of packets to mark corrupted in the buffer belonging to $s$
- $d$ is a set of indices of packets to remove from the buffer belonging to $s$

**Buffers**

We define **buffers$_{N,B}$** to be an element of

$$((\text{Packet} \cup \text{Null}) \times \mathbb{N}_{B+1} \times \mathcal{P}(\mathbb{N}_B) \times \mathcal{P}(\mathbb{N}_B))^{2N} \overset{\text{df}}{\to} (\text{Packet} \cup \text{Null})^{2N}$$

such that:

- it contains state to store $2N$ buffers of capacity $B$
- when called with a list of $(p_{\text{snd}}, i, m, d)_s$ as defined above, it performs the operations on the relevant buffers, and returns a list of packets to deliver $(p_{\text{rcv}})_s$

## §B.2.2 Sender

We define **Sender$_L$** to be the set

$$(\text{Packet} \cup \text{Null}) \times (\text{Byte} \cup \text{Null}) \times \text{Bool} \times \mathbb{N}_L \overset{\text{df}}{\to} (\text{Packet} \cup \text{Null}) \times (\text{Byte} \cup \text{Null}) \times \text{Bool}$$

When it is called, a sender receives $(p'_{\text{rcv}}, b_i, n_o, c)$ where

- $p'_{\text{rcv}}$ is the packet that was delivered by the network last step (if it exists)
- $b_i$ is the next byte in the input stream (if available)
- $n_o$ indicates whether the byte returned in the last call to $F$ has been consumed by the traffic behaviour
- $c$ is the size of the congestion window

The function $F$ is then applied to the resulting state to give $(p_{\text{snd}},\ b_o,\ n_i)$ where

- $p_{\text{snd}}$ (if it exists) is the packet that should be pushed to the buffer of the opposite sender in this timestep,
- $b_o$ is the next byte in the output stream (if available),

- $n_i$ indicates whether $b_i$ was "ingested" by the sender (i.e. whether the next byte in the input stream should be provided in the next round)

If the sender is a real TCP sender, we would expect that:

$$\begin{aligned} &\mathrm{null}(p_{\mathrm{snd}}) \;\vee\; p_{\mathrm{snd}} \in \mathrm{sent} \\ \vee\;\; &\mathrm{SeqNum}(p_{\mathrm{snd}}) = \mathrm{last\_seq} + 1 < \mathrm{window\_start} + c \end{aligned}$$

where

$$\mathrm{last\_seq} = \begin{cases} \max_{p\,\in\,\mathrm{sent}}(\mathrm{SeqNum}(p)) & \text{if sent} \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

and

$$\mathrm{window\_start} = \begin{cases} \min_{p\,\in\,\mathrm{sent}\,\setminus\,\mathrm{ack\_received}} \mathrm{SeqNum}(p) & \text{if } (\mathrm{sent} \setminus \mathrm{ack\_received}) \neq \emptyset \\ \mathrm{last\_seq} + 1 & \text{otherwise.} \end{cases}$$

By inspecting the behaviour of $(b_o, n_o)$ at each timestep, we can determined the stream of bytes previously output by sender $s$ after $t$ steps $\mathrm{Output}(s,t)$. We can apply a similar strategy to with $(b_i, n_i)$ to get $\mathrm{Input}(s,t)$. If the sender is a real TCP sender, we would also expect that, for some $l \in \mathbb{N}$, all simulations satisfy:

$$\forall (s,s') \in \mathrm{uniflows}.\ \forall t \in \mathbb{N}.\ \exists u \in \mathrm{Byte}^*.\ \mathrm{Input}(s,t) = \mathrm{Output}(s',t)u \wedge |u| \leq l$$

### CCA

We define **$\mathbf{CCA}_L$** to be the set

$$(\mathrm{SeqNum} \cup \mathrm{Null}) \times (\mathrm{AckNum} \cup \mathrm{Null}) \xrightarrow{\mathrm{df}} \mathbb{N}_L$$

This function is called before the sender with $\mathrm{seqnum\_or\_null}(p'_{\mathrm{snd}})$ and $\mathrm{acknum\_or\_null}(p'_{\mathrm{rcv}})$.

### Traffic Behaviour

We define **$\mathbf{TrafficBehaviour}_N$** to be the set

$$((\mathrm{Byte} \cup \mathrm{Null}) \times \mathrm{Bool})^{2N} \xrightarrow{\mathrm{pi}} ((\mathrm{Byte} \cup \mathrm{Null}) \times \mathrm{Bool})^{2N}$$

At each step, this is called with a list of $(b'_o, n'_i)_s$ of the previous step, and returns a list of $(b_i, n_o)_s$ for the current step.

### Network

We define a **$\mathbf{Network}_L$** as a tuple

$$(N, B, \mathrm{interconnect}, \mathrm{sender}, \mathrm{traffic})$$

where

- $N \in \mathbb{N}$ is the number of bidirectional flows
- $B \in \mathbb{N}$ is the limit on the number of packets in transit for a given unidirectional flow

- network : $\text{InterconnectBehaviour}_{N,B}$
- sender : $\text{Sender}_L$
- traffic : $\text{TrafficBehaviour}_N$

These components be arranged into a "steppable" simulation of type $() \xrightarrow{\text{pi}} ()$ – see Figure 11.

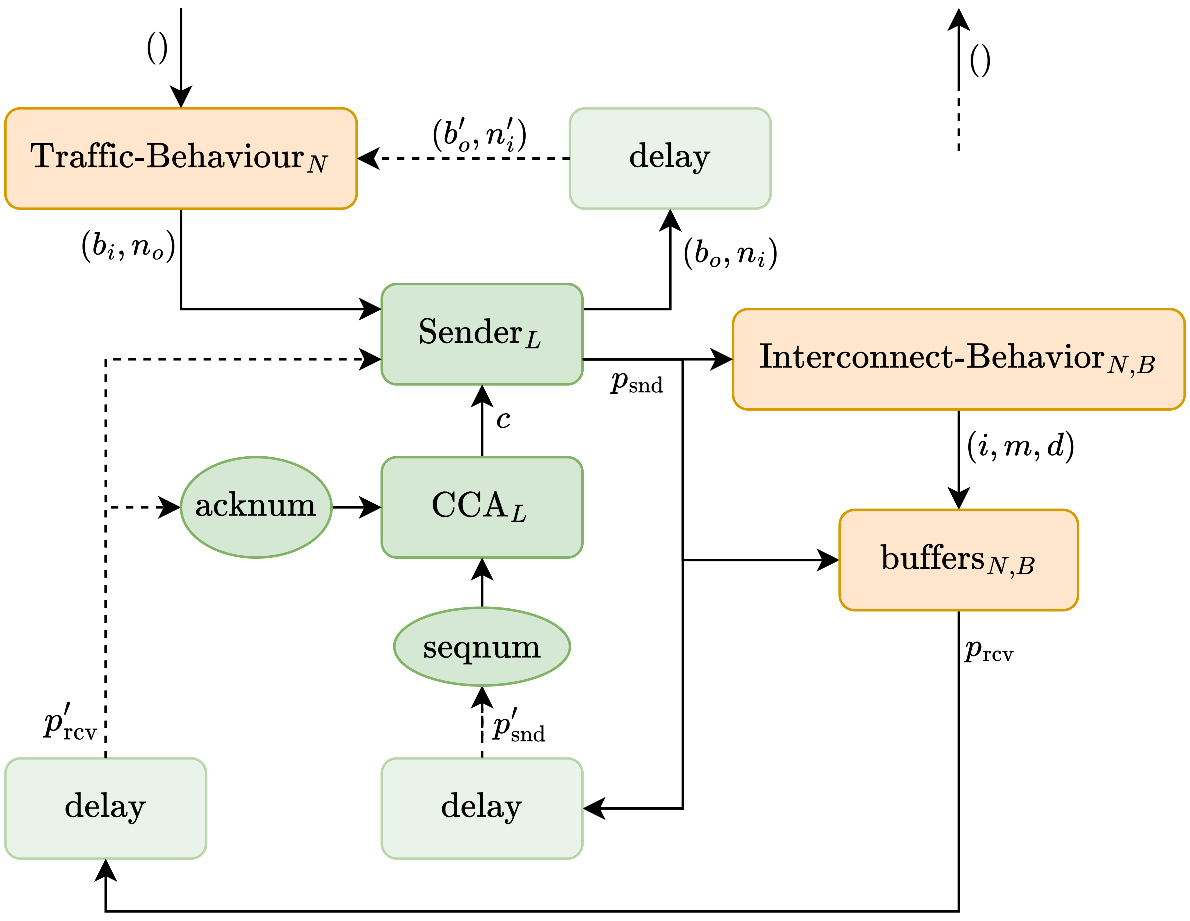


**Figure 11**: Diagram showing how components can be arranged into a steppable simulation of type $() \xrightarrow{\text{pi}} ()$. "Delay" commponents store their input, and emit it the next time they are called. Connections for a single sender are shown – for more senders, the components in green are replicated, with all CCAs and senders being called in parallel.

## §B.3 Continuous AD′-POMDP

We can construct a continuous AD′-POMDP (Section 2.2.2) from a tuple $(T, D, u, \alpha)$ where

- $T$ is a CCA-Template $(\mathcal{O}', \mathcal{A}', F, L)$ as defined in Section 2.2.3,
- $D$ is a probability distribution over $\text{Network}_L$,
- $u$ : Utility is a utility function, and
- $\alpha \in \mathbb{R}$ is a discount rate.

We do this by constructing a discrete (i.e. regular) AD'-POMDP for some arbitrary number of steps-per-second $g$:

- $\mathcal{N} = \{1, 2, ..., 2N\}$ where $N$ is the maximum number of flows that a network $n$ drawn from $D$ can have
- $\mathcal{O} = (\mathcal{O}')^{2N}$
- $\mathcal{A} = (\mathcal{A}')^{2N}$
- $\gamma = e^{-\alpha/g}$

$S, \Omega, U, P, R$ are defined implicitly through a simulation model:

**Algorithm 4:** Discrete AD'-POMDP for given $g$

```
1 Randomly draw a Network from $D$, and create a simulation using it. The
    simulation should use $F(\pi')$ as the CCA, for some placeholder policy $\pi'$.
2 loop
3     Let $n$ be the current number of steps completed. Let $t = n/g$.
4     Calculate utility at the current time $u_t$.
5     Interpret utility as a "rate of reward", and calculate reward as $r_n = u_t/g$.
6     Step the simulation. Let the set of available agents for the current state be
        given as $A = \{i \mid \pi'_i$ is called with $o_i$ during the step$\}$. Intercept these $\pi'$
        calls.
7     Yield active senders $A$, observations $\langle o_i \mid i \in A_i \rangle$, and reward $r_n$.
8     Resume with actions $\langle a_i \mid i \in A_i \rangle$. Give the output of each intercepted $\pi'_i$
        call as $a_i$, and finish the step.
9 end loop
```

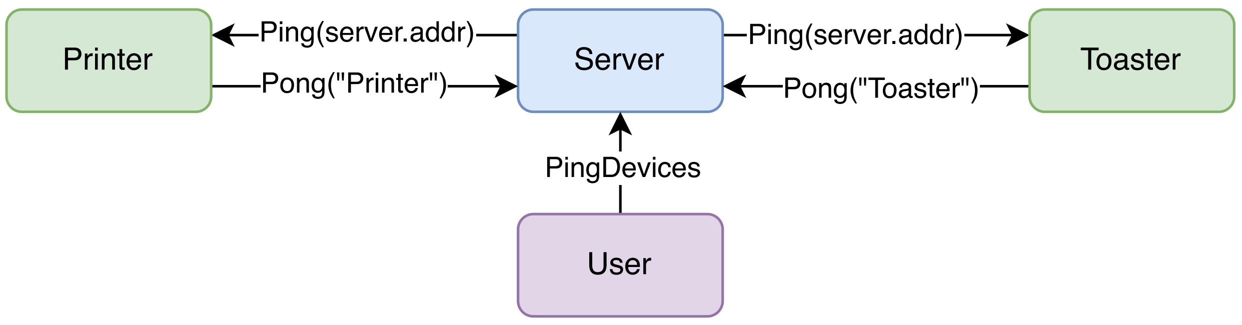
With this discrete AD'-POMDP, we can substitute $n = gt$, $\gamma = e^{-\alpha/g}$, and $r_n = u_t/g$ into Equation (2.1) to get

$$R(\tau) = \sum_{n=0}^{|\tau|-1} \left(e^{-\alpha/g}\right)^n \frac{u_t}{g} = \sum_{n=0}^{|\tau|-1} \frac{e^{-\alpha t} u_t}{g}$$

We take $g \to \infty$ to get a continuous AD'-POMDP. If we have an infinite-length trajectory, treating this as a Riemann sum with $\Delta = 1/g$ gives Equation (2.4) where $r_t = u_t$.

# C Simulation example

In this chapter, we explore how a simple simulation can be built using the `flowforge::simulation` module. We consider a scenario where a user periodically instructs a server to ping devices on a network.



```
use derive_more::{From, TryInto};
use derive_where::derive_where;
use flowforge::{
    quantities::{seconds, Time},
    simulation::{Address, Component, Message, SimulatorBuilder},
    util::{logging::NothingLogger, never::Never},
};
use generativity::make_guard;

struct Ping<'sim, E> {
    source: Address<'sim, Pong, E>,
}
struct Pong {
    name: String,
}

#[derive(Debug)]
struct Printer;

impl<'sim, E> Component<'sim, E> for Printer {
    type Receive = Ping<'sim, E>;

    fn receive(&mut self, p: Ping<'sim, E>, _: Time) -> Vec<Message<'sim, E>> {
        vec![p.source.create_message(Pong {
            name: "Printer".into(),
        })]
    }
}
```

```rust
#[derive(Debug)]
struct Toaster;

impl<'sim, E> Component<'sim, E> for Toaster {
    type Receive = Ping<'sim, E>;

    fn receive(&mut self, p: Ping<'sim, E>, _: Time) -> Vec<Message<'sim, E>> {
        vec![p.source.create_message(Pong {
            name: "Toaster".into(),
        })]
    }
}

#[derive_where(Debug)]
struct Server<'sim, E> {
    address: Address<'sim, ServerMessage, E>,
    devices: Vec<Address<'sim, Ping<'sim, E>, E>>,
}

struct PingDevices;

#[derive(From, TryInto)]
enum ServerMessage {
    SendPing(PingDevices),
    Pong(Pong),
}

impl<'sim, E> Component<'sim, E> for Server<'sim, E>
where
    E: 'sim,
{
    type Receive = ServerMessage;

    fn receive(&mut self, e: ServerMessage, time: Time) -> Vec<Message<'sim, E>> {
        match e {
            ServerMessage::SendPing(PingDevices) => {
                println!("Sent pings at time {time}");
                self.devices
                    .iter()
                    .map(|address| {
                        address.create_message(Ping {
                            source: self.address.clone().cast(),
                        })
                    })
                    .collect()
            }
            ServerMessage::Pong(Pong { name }) => {
                println!("Received pong from {name} at time {time}");
                vec![]
            }
        }
    }
}

#[derive_where(Debug)]
struct User<'sim, E> {
    next_send: Time,
    server: Address<'sim, PingDevices, E>,
}

impl<'sim, E> Component<'sim, E> for User<'sim, E> {
    type Receive = Never;

    fn next_tick(&self, _: Time) -> Option<Time> {
        Some(self.next_send)
    }

    fn tick(&mut self, _: Time) -> Vec<Message<'sim, E>> {
        self.next_send = self.next_send + seconds(1.);
        vec![self.server.create_message(PingDevices)]
    }
}

#[derive(From, TryInto)]
enum GlobalMessage<'sim> {
    Server(ServerMessage),
    Ping(Ping<'sim, GlobalMessage<'sim>>),
    Never(Never),
}

#[allow(unused_variables)]
fn main() {
    make_guard!(guard);
    let builder = SimulatorBuilder::<GlobalMessage>::new(guard);

    let printer_address = builder.insert(Printer);
    let toaster_address = builder.insert(Toaster);

    let server_slot = builder.reserve_slot();
    let server = Server {
        address: server_slot.address(),
        devices: vec![printer_address, toaster_address],
    };
    let server_address = server_slot.fill(server);

    // We can't just cast addresses to any type (well-typed communication)

    // builder.insert(User {
    //     next_send: Time::SIM_START,
    //     server: printer_address.cast(),
    // });
    // COMPILE ERROR - expected Address<PingDevices, _>, found Address<Ping, _>
    builder.insert(User {
        next_send: Time::SIM_START,
        server: server_address.cast(),
    });

    // Attempting to create a simulator with the same lifetime doesn't work...

    // let builder = SimulatorBuilder::<GlobalMessage>::new(guard);
    // COMPILE ERROR - guard already moved

    // let builder = SimulatorBuilder::<GlobalMessage>::new(Guard::new( ... ));
    // COMPILE ERROR - unsafe

    // ... but we can create one with a different lifetime.
    make_guard!(guard);
    let builder2 = SimulatorBuilder::<GlobalMessage>::new(guard);

    // Since the simulators have different 'sim lifetimes, we can't use an address
    // from sim1 with builder2 (simulator isolation)

    // builder2.insert(User {
    //     next_send: Time::SIM_START,
    //     server: server_address.cast()
    // }); COMPILE ERROR - invariant lifetime clash

    let mut sim = builder.build(NothingLogger).unwrap();
    while sim.time() < Time::from_sim_start(seconds(3.)) && sim.tick() {}

    // Sent pings at time 0.00st
    // Received pong from Printer at time 0.00st
    // Received pong from Toaster at time 0.00st
    // Sent pings at time 1.00st
    // Received pong from Printer at time 1.00st
    // Received pong from Toaster at time 1.00st
    // Sent pings at time 2.00st
    // Received pong from Printer at time 2.00st
    // Received pong from Toaster at time 2.00st
}
```

# D MAPPO overview

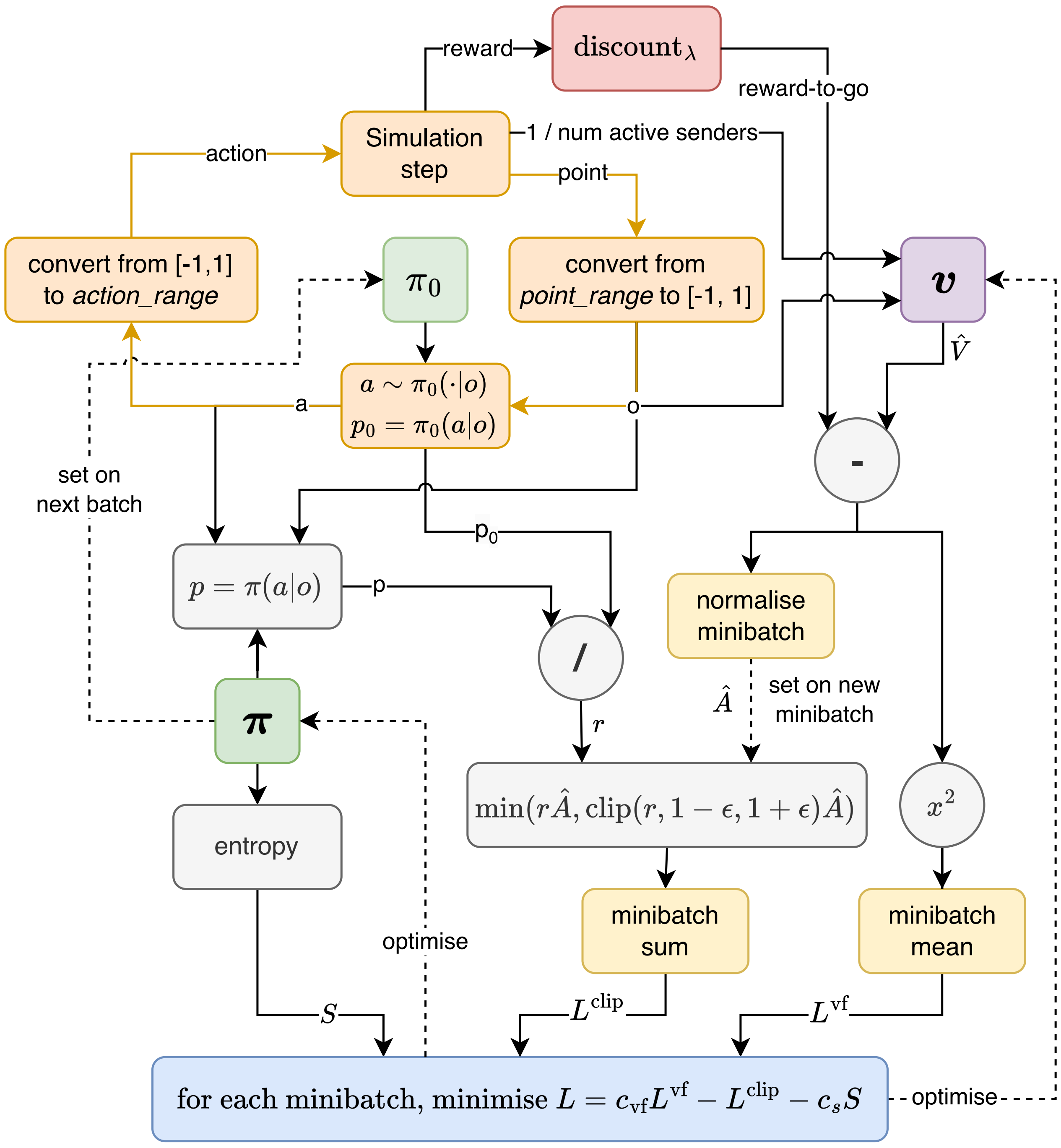


**Figure 12**: Overview of the MAPPO variant used by Remyr.

# E Configuration files

## §E.1 Default network

```
{
  "Remy": {
    "rtt": {
      "type": "uniform",
      "min": "100ms",
      "max": "200ms"
    },
    "bandwidth": {
      "type": "uniform",
      "min": "10Mb/s",
      "max": "20Mb/s"
    },
    "loss_rate": {
      "type": "always",
      "value": 0.0
    },
    "buffer_size": null,
    "num_senders": {
      "type": "uniform",
      "min": 1,
      "max": 16
    },
    "off_time": {
      "type": "exponential",
      "mean": "5s"
    },
    "on_time": {
      "type": "exponential",
      "mean": "5s"
    }
  }
}
```

## §E.2 Proportional throughput-delay fairness

```
{
  "AlphaFairness": {
    "alpha": 1.0,
    "beta": 1.0,
    "delta": 1.0,
    "worst_case_rtt": "10s"
  }
}
```

## §E.3 Default Remy trainer

```
{
  "type": "remy",
  "rule_splits": 12,
  "optimization_rounds_per_split": 2,
  "min_action": {
    "window_multiplier": 0.0,
    "window_increment": 0,
    "intersend_delay": "0.25ms"
  },
  "max_action": {
    "window_multiplier": 1.0,
    "window_increment": 256,
    "intersend_delay": "3ms"
  },
  "initial_action_change": {
    "window_multiplier": 0.01,
    "window_increment": 1,
    "intersend_delay": "0.05ms"
  },
  "max_action_change": {
    "window_multiplier": 0.5,
    "window_increment": 32,
    "intersend_delay": "1ms"
  },
  "action_change_multiplier": 4,
  "default_action": {
    "window_multiplier": 1.0,
    "window_increment": 1,
    "intersend_delay": "3ms"
  },
  "change_eval_config": {
    "network_samples": 50,
    "run_sim_for": "60s"
  },
  "count_rule_usage_config": {
    "network_samples": 500,
    "run_sim_for": "60s"
  },
  "drill_down": true
}
```

## §E.4 Default Remyr trainer

```
{
  "type": "remyr",
  "iters": 2000,
  "updates_per_iter": 5,
  "num_minibatches": 4,
  "min_point": {
    "ack_ewma": "0ms",
    "send_ewma": "0ms",
    "rtt_ratio": 1.0
  },
  "max_point": {
    "ack_ewma": "500ms",
    "send_ewma": "500ms",
    "rtt_ratio": 5.0
  },
  "min_action": {
    "window_multiplier": 0.0,
    "window_increment": 0,
    "intersend_delay": "0.25ms"
  },
  "max_action": {
    "window_multiplier": 1.0,
    "window_increment": 256,
    "intersend_delay": "3ms"
  },
  "hidden_layers": [
    32,
    16
  ],
  "entropy_coefficient": 0.01,
  "value_function_coefficient": 0.5,
  "learning_rate": 0.0003,
  "learning_rate_annealing": true,
  "clip": 0.2,
  "clip_annealing": true,
  "weight_decay": null,
  "discounting_mode": {
    "type": "continuous_rate",
    "half_life": "1s"
  },
  "bandwidth_half_life": "100ms",
  "rollout_config": {
    "network_samples": 100,
    "run_sim_for": "60s"
  },
  "repeat_actions": {
    "type": "uniform",
    "min": 0,
    "max": 200
  }
}
```

# F Project Proposal

Title: TCP ex Machina

## Introduction

### TCP ex Machina

Congestion control algorithms are used in networks to control the rate at which senders send data, with the aim of maximising "performance" (usually some metric formed by a combination of throughput and latency). They define how nodes measure the conditions of the network, and how they act in response to these measurements. Traditionally, such congestion control algorithms have been developed using a by hand, "one size fits all" approach, with a single algorithm being used across a wide variety of networks and environments. However, these human designed algorithms contain implicit assumptions about the network conditions the nodes are operating in. When network conditions deviate from these assumptions, performance often suffers, and the emergent behaviour of such a system can be unpredictable.

One research project that explored an alternative is TCP ex Machina (Winstein and Balakrishnan). The researchers proposed a generic, customisable congestion control algorithm, and a C++ program called "Remy" that tailors the algorithm to a given set of network assumptions. However, the public implementation of Remy has become neglected, with the last commit over five years ago, and issues such as the lack of license preventing more widespread use.

### Rust

Rust is a modern programming language with an emphasis on performance, type safety, and concurrency. It uses a number of compile time techniques (such as lifetimes, the borrow checker, and send & sync traits) to guarantee that code which uses the safe portion of the language is free from data races, segfaults, double frees, and other classes of error common in systems languages like C++. Because of these guarantees, many Rust projects prefer to use pure Rust dependencies (rather than those with bindings to other languages), leading to higher demand for pure implementations of libraries and frameworks.

## Reinforcement Learning

Reinforcement learning is an area of machine learning that focuses on how virtual agents can learn to maximise a reward function by simply interacting with an environment and receiving feedback on the reward. A simple approach to such problems is Q-learning, in which the agent attempts to learn the expected reward for a given action taken in a given state (and hence the best action can be found as the action with the highest expected reward for the current state). This approach has been modified to form techniques such as Deep Q-Learning (DQN), as well as many other more modern techniques.

## Project Description

The core of this project will involve attempting to replicate the Remy program described in TCP ex Machina using Rust. It is a non-goal to support the "poisson" version present in the current Remy GitHub repository.

Core criteria:

- The program should be written primarily in the Rust programming language.
- The program should create congestion control algorithms that perform similarly to the original Remy implementation when compared to other congestion control algorithms, unless there is clear justification for why it shouldn't (e.g. if there is a discrepancy between the algorithm described in the paper and the original implementation). This criteria may be evaluated by extending the already modified ns-2 network simulator used in the original paper, and then comparing characteristics such as the throughput / delay distribution across simulation runs for one or more network / traffic models.
- The program should be architected in a way that makes it easy to add support for other potential mechanisms for learning congestion control algorithms. This criteria can be evaluated qualitatively via a written analysis of the architecture, and potentially further supported by implementing one of the extensions.

Extension criteria:

- The program should include a mode that allows a congestion control algorithm (likely of a different structure to that described in the TCP ex Machina paper) to be trained using reinforcement learning. This approach should then be investigated by comparing both the computational requirements required for training the two approaches, as well as by comparing the performance of the algorithms that each mode creates.
- The program should include a mode that allows a genetic algorithm to be used as an addition to or a replacement for the original mechanism for training the congestion control algorithm. This approach should be investigated in a similar manner to the above extension.

- The program should support a wider range of network configurations than Remy. The project should then seek to evaluate the effectiveness of the program in one or more examples of these extended network configurations.

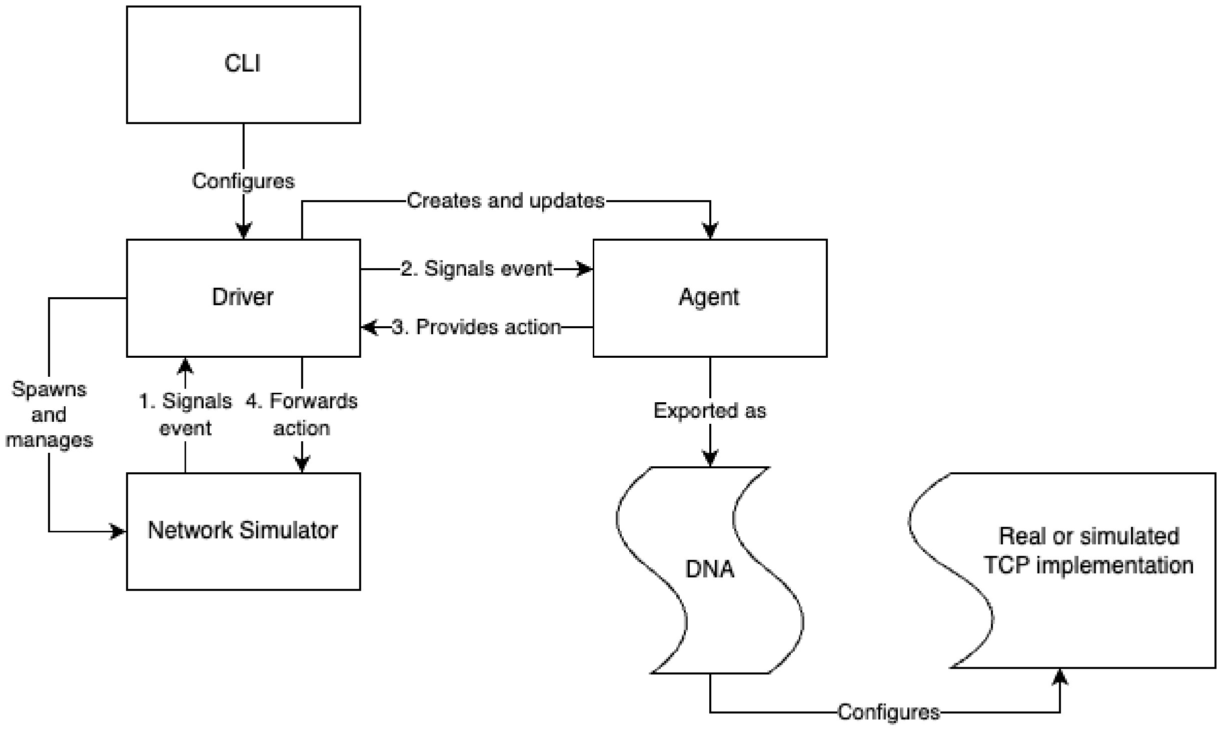
## Substance and Structure

The following section presents a high-level draft for how the project might be structured. This architecture is likely to be changed during the early stages of the project.

There are 5 main parts that need to be created or configured:

- CLI (responsible for parsing configuration and starting the driver)
- Driver (implements the training algorithm for a given type of agent)
- Agent (an instance of a congestion control algorithm, described by DNA)
- DNA (describes the behaviour of an agent in a binary format, e.g. protobuf)
- Network Simulator (likely to be a custom, lightweight discrete event simulator)

Below is a diagram showing how these parts interact with each other, and with other previously created parts. The numbered arrows describe the core event loop during a simulation.



Doing the genetic extension will likely require only the existing driver to be extended, whilst the RL extension may require an entirely new driver, agent, and DNA format. The RL and network architecture extensions will also require changing the network simulator component.

## Starting Point

As of the submission of this proposal:

- I have slightly modified the existing Remy implementation and the TCP ex Machina reproduction code code to ensure they compile and run on my machine. I may further modify the aforementioned reproduction code to support CC algorithms created using my version of Remy, for use in debugging and the evaluation section of the dissertation.

## Resource Declaration

### Physical Resources

- Personal MacBook Air (M1, 16GB, 512GB) for general development and dissertation writing
- AWS EC2 instance OR personal desktop PC (Ryzen 2600x, 16GB, RTX 2060, 2TB) for x64 only software or long-running tasks (e.g. training, evaluation)

### Software Resources

The project is largely contingent on the original Remy software and reproduction code working as expected, as it will use both of these in establishing baseline performance for the new program. So far I have managed to get both of these parts working in isolation, but will need to check that they can be combined.

Other software resources are likely to be freely available online.

### Contingency Plans

Source code for the project and dissertation will be version controlled using Git and GitHub, and backed up regularly both locally and to the cloud (likely OneDrive).

Should my laptop fail, I should be able to use the desktop, and vice versa (albeit with decreased performance when running x64 programs). As a worst case failsafe, I should be able to remotely develop using an AWS EC2 instance.

*I accept full responsibility for these machines and I have made contingency plans to protect myself against hardware and/or software failure.*

## Project Plan

The project plan is divided into two-week slots from the start of term until the dissertation deadline. Notes are marked by *italic*.

| **Start Date** | **Planned Work** | **Concrete Deliverables** |
|---|---|---|
| 02/10/2023 | Finalise and submit proposal, ensure that the original Remy program and the modified ns-2 simulator run correctly, research tools and techniques that may be useful for the project. | Project proposal. |
| 16/10/2023 | Setup Rust project, GitHub repo, and create mock components for the whole system. Start work on the lightweight network simulator. | CLI that can generate random (dumbell) networks according to configuration and mocks the rest of the core functionality. |
| 30/10/2023 | Finish network simulator & accompanying unit tests, and start work on agent and driver. | Simulation of simple agents sending data on a dumbell network, should output relevant metrics / graphs. |
| 13/11/2023 | *Upcoming module deadline, so reduced productivity expected.* Continue work on agent and driver. | Remy training mode, including a graph of algorithm performance against iterations. |
| 27/11/2023 | *Module deadline on 29/11/2023.* Finish agent and driver (and accompanying unit tests), and perform a mini evaluation by comparing against existing implementation. | Remy DNA export, along with integration into full network simulator. Small report comparing the program against other CC algorithms. |
| 11/12/2023 | Contingency. | Core deliverable that should satisfy core criteria. |
| 25/12/2023 | Holiday. | |
| 08/01/2024 | Begin work on extensions. Perform any relevant research not yet completed, and potentially start experimenting with a proof of concept for the technique. | Small report on the feasibility of one or more extensions, including relevant prototypes or demonstrations. |
| 22/01/2024 | Work on progress report, including evaluation of the core deliverable. | Progress report, including evaluation of the core deliverable (deadline 02/02/2024). |

| | | |
|---|---|---|
| 05/02/2024 | Present progress report sometime between 7th and 14th. Attend dissertation briefing. Continue implementing extensions. | Progress report presentation. Program implementing the main aspect of the extension. |
| 19/02/2024 | I should aim to finish implementation of extensions by the end of this slot. | Integration of the extension into the network simulator (if applicable). Small report evaluating the extension. |
| 04/03/2024 | Work on project evaluation. I should finish the deliverable, including extensions, by the end of this slot. | Full deliverable. |
| 18/03/2024 | Start dissertation by writing introduction and then the research / preparation parts. | Introduction and research / preparation sections. |
| 01/04/2024 | Write the implementation and evaluation sections. Should aim to finish first draft. | First draft of dissertation. |
| 15/04/2024 | Gather feedback, iterate, finalise, and ideally submit dissertation. | Second draft of dissertation. |
| 29/04/2024 | Contingency. | Finished dissertation (deadline 10/05/2024). |