CLEAVE Paper for CNERT'21

Manuel Olguín Muñoz
School of Electrical Engineering
and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden

James Gross
School of Electrical Engineering
and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden

***Abstract*—**

TODO

I. Introduction

II. Approach

Existing platforms for NCS benchmarking take one of two approaches. The first is a fully simulated approach, in which the plant, network and controller reside in a completely simulated environment. This allows for unparalleled flexibility, as parts of the system can be swapped out or modified without greater difficulty. However, this comes at the cost of realism, as these models are always approximations of reality; and time, as these simulations are often very computationally expensive.

The alternative is an approach using a testbed comprised of a *physical* plant that interacts with a *real* network and controller. This of course allows for great realism, but provides very little flexibility in terms of the components of the system, limiting the approach to whatever plant, network and controller are available.

CLEAVE tries to mitigate the shortcomings of these approaches by walking the line between them. We employ a third strategy based on having a real-time emulated representation of a control system interacting with a real network. This allows for great flexibility when it comes to the components of the control system, as these are as easily switched out as in the fully simulated approach, while maintaining the realism of arguably the most complex and limiting component of an NCS, namely the network itself.

In the following we will explain the approach taken in CLEAVE by walking the reader through the steps necessary to deploy an arbitrary NCS emulation on top of the framework. An overview of the structure of an NCS emulation on CLEAVE can be seen in fig. 1.

A. Defining a Plant

The first step in developing an NCS emulation on top of CLEAVE is the definition of a Plant which represents the physical system to be controlled.

In abstract terms, Plants in CLEAVE can be understood to be composed of 1) an emulation of a physical system which is updated in a discrete-time fashion; 2) procedures to modify or act upon the physical emulation at each time-step; 3) procedures to sample relevant values from the physical emulation at each time-step; and 4) procedures to interact with the network and send and receive data from the Controller.

While item 4 is handled transparently by CLEAVE, by hiding it through layers of abstraction, the rest of the items need to be provided in more explicit ways by the user.

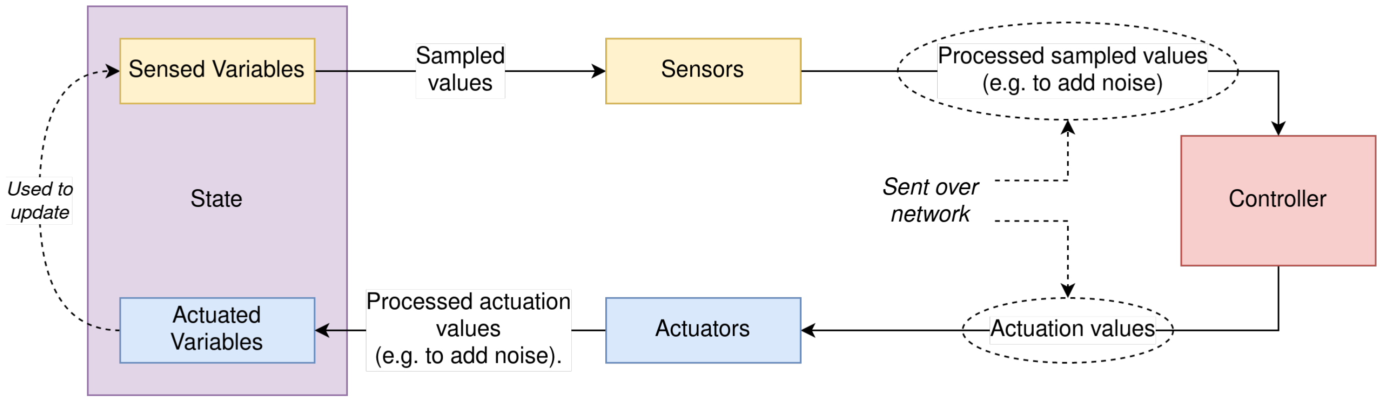

Fig. 1: Structure of an emulated NCS on top of CLEAVE. The State object implements the discrete-time evolution of the Plant. For this purpose, it holds two sets of special variables; Actuated Variables and Sensed Variables. Actuated Variables are updated from the values obtained from the Actuator objects at the beginning of each time step. These are used to perform a time step update in the State, and update the values of the Sensed Variables. The values of the Sensed Variables are then, at the end of each time step, sampled and passed on to the Sensor objects, which in turn process them (for instance, to add noise) and send the processed values to the Controller. The Controller uses these values to calculate the new actuation commands and sends these to the Actuators. Finally, the Actuators process the actuation commands and hold the new values for the Actuated Variables until they are read at the next time step.

For item 1, the user needs to extend an abstract base class `State` provided by the framework. This class provides the base functionality and defines a simple API for the implementation of discrete-time emulations by 1) automatically tracking variables that can be modified and/or sampled by the framework; and 2) defining a single abstract method `State.advance()` which is invoked by the framework on each time step of the emulation.

Items 2 and 3 are in the same way provided by the user through the extension of the `Actuator` and `Sensor` abstract base classes, respectively. `Actuator` objects process the outputs of the Controller and update the values of the `State` accordingly. Conversely `Sensor` objects sample relevant values from the `State` object, process them, and finally pass them on to the Controller through the network.

B. *Defining a Controller*

After the Plant has been defined, users need to implement a Controller for it. This is once again done by extending an abstract base class, `Controller`, provided by CLEAVE. This class defines a single abstract method, `Controller.process()` which is called by the framework whenever samples are provided by the Plant. When invoked, it is passed a mapping of property names to sampled values. The framework also expects implementations of this method to return yet another mapping, this time of actuated property names to target values. Users are otherwise free to implement any kind of logic or process within extending classes.

C. *Configuring and running the emulation*

Once the core components for the emulation have been defined and implemented, the emulation needs to be configured to register them with the framework. This is done through fully-featured Python scripts which simply define a number of required (and some optional) top-level variables. These Python configuration files are then passed as arguments to the `cleave` bash command provided by the framework, which sets up and executes the emulation. The use of Python scripts for configuration allows for a high degree of flexibility and complexity in the definition of emulation setups.

At this point, yet another core component of CLEAVE needs to be mentioned: the Dispatcher server. Although Controllers in CLEAVE can be started manually, this quickly becomes cumbersome when performing experimentation with more than a few Plants. The CLEAVE Dispatcher server solves this by providing a way for remote deployment of Controllers through a RESTful HTTP service that the Plant connects to (see fig. 2).

Example configuration files for a Plant and the Dispatcher are presented in listings 1 and 2.

III. Experimental validation

WIP

In this section, we validate the utility of the CLEAVE framework through scalability measurements of a networked control system running on a WiFi link. We aim to answer questions about the ability of CLEAVE to provide relevant and accurate metrics on the performance of such setups as *system load* increases. In this context, we will focus on the load as it manifests on the network link in terms of the reliability of the link and the associated latencies. We adopt this constrained view of load due to the characteristics of NCS, where controllers are usually relatively lightweight in terms of computing resources, leaving the network link as the principal bottleneck in the system.

```python
from cleave.api.plant import
↪  SimpleConstantActuator, SimpleSensor
from cleave.impl.inverted_pendulum import
↪  InvPendulumState

host = '0.0.0.0'  # address of dispatch server
port = 8080       # providing Controllers

# Controller parameters
controller_class = 'InvPendulumController'
controller_params = {'ref': 0.0}

# Plant emulation parameters
tick_rate = 100
emu_duration = '30m'
state = InvPendulumState(fail_angle_rad=0.34)
output_dir = './plant_metrics'

# Sensors to attach to the Plant
sensors = [
    SimpleSensor('position', 100),
    SimpleSensor('speed', 100),
    SimpleSensor('angle', 100),
    SimpleSensor('ang_vel', 100),
]

actuators = [
    SimpleConstantActuator(
        initial_value=0,
        prop_name='force')
]
```

Listing 1: Example configuration file for a CLEAVE Plant, defining the parameters for a single emulation. The "controller parameters" define the class of Controller to request from the Dispatcher. "Plant emulation parameters" define key emulation parameters, such as the emulation discrete-time update rate (the *tick rate*). The Sensors and Actuators are then provided in lists, each Sensor and Actuator attached to a specific property identified by a string. Additionally, Sensors have a sampling rate in Hz which needs to be provided as an integer.

```python
from cleave.impl.inverted_pendulum import
↪  InvPendulumController

host = 'localhost'
port = 8080

controllers = {
    'InvPendulumController':
    ↪  InvPendulumController,
}

output_dir = './controllers'
```

Listing 2: Example configuration file for the CLEAVE Dispatcher. The previously defined Controller is registered with the framework and will then be available from the Dispatcher server for use with a Plant.

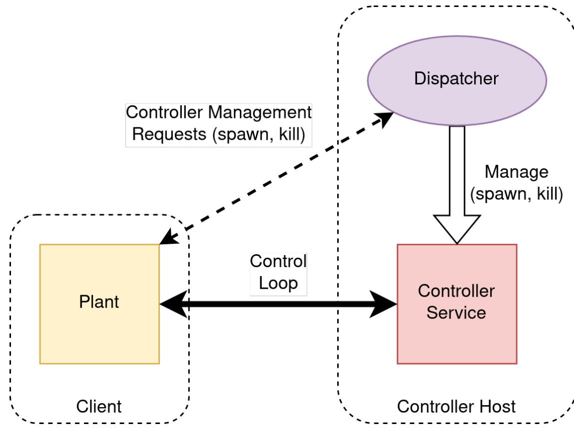

Fig. 2: Diagram of the interaction between CLEAVE Plant, Dispatcher and Controller. The Dispatcher is instantiated and listens on a specific port on the Controller Host. As the Plant initializes its emulation, it requests a Controller from the Dispatcher; the Dispatcher spawns the corresponding Controller and returns the host and port on which it is listening. The Plant proceeds to execute its emulation configuration. Finally, on shutdown, the Plant once again communicates with the Dispatcher to request the shutdown of the associated Controller.

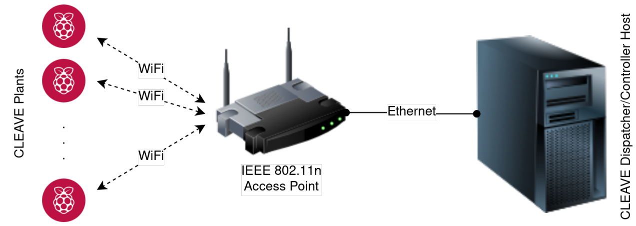

Fig. 3: Experiment setup

Our experimental setup is depicted in fig. 3. A number between 1 and 12 of CLEAVE Plants running on dedicated light-weight general purpose computing devices (in this case, Raspberry Pi 4's) connect wirelessly to an 802.11n WiFi Access Point (AP). This AP in turn connects via Ethernet to a host on which the CLEAVE Dispatcher and Controllers are executed.

In terms of the control system deployed on top of CLEAVE for the experiments, we chose a two-dimensional inverted pendulum system. This system was chosen due to its relative simplicity and prevalence in the field of automatic control as one of the fundamental examples basically every control engineer has encountered at some point.

The physical system, represented in fig. 4, was implemented using CLEAVE's API and a 2D physics library [1, 2]. For the controller, a proportional-differential strategy was employed, implemented using the framework Controller API and the NumPy numeric computation library [3].

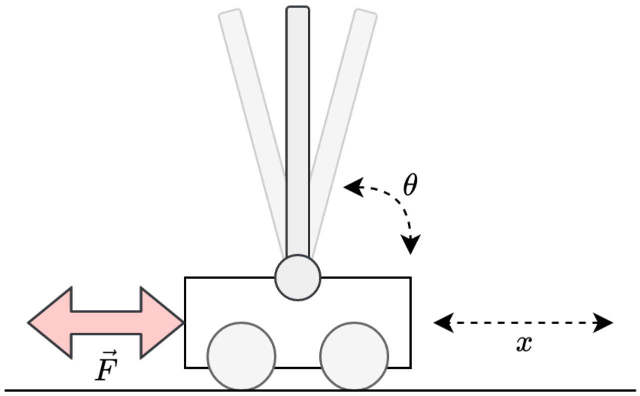

Fig. 4: The two-dimensional inverted pendulum system. The cart moves on the X-axis, and the pendulum on top of it swings freely. The objective of the system is to balance the pendulum vertically through the application of horizontal forces on the cart.

Acknowledgements

References

[1] *Chipmunk2D Physics*. [Online]. Available: https://chipmunk-physics.net/.

[2] *Pymunk*. [Online]. Available: http://www.pymunk.org/en/latest/.

[3] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.