A seminar on Discrete-Event Simulation (DES) with SimPy

for the Computational Models for Complex Systems 24/25 course

by Federico Cioni

Seminar outline

1. DES introduction
2. SimPy introduction
3. Bus allocation scenario

1 Discrete-Event Simulation

Discrete-Event Simulation

Discrete Event Simulation is a technique for modeling the behavior of a system as a sequence of events that occur at discrete points in time.

A DES starts in a certain State, which is a description of a system configuration in terms of a set of variables.
A state can be updated by Events, that can be istantaneous or may trigger an Activity, a process that has a duration (e.g. customer is being served).

Discrete-Event Simulation

Additional components:

- **Future Event List (FEL): When a new event is created (e.g. a costumer joins a queue) it is added to a FEL. it is a list of events that are pending as a result of previously simulated event but have yet to be simulated themselves.**
- **Clock: The simulation must keep track of the current simulation time, in whatever measurement units are suitable for the system being modeled.**

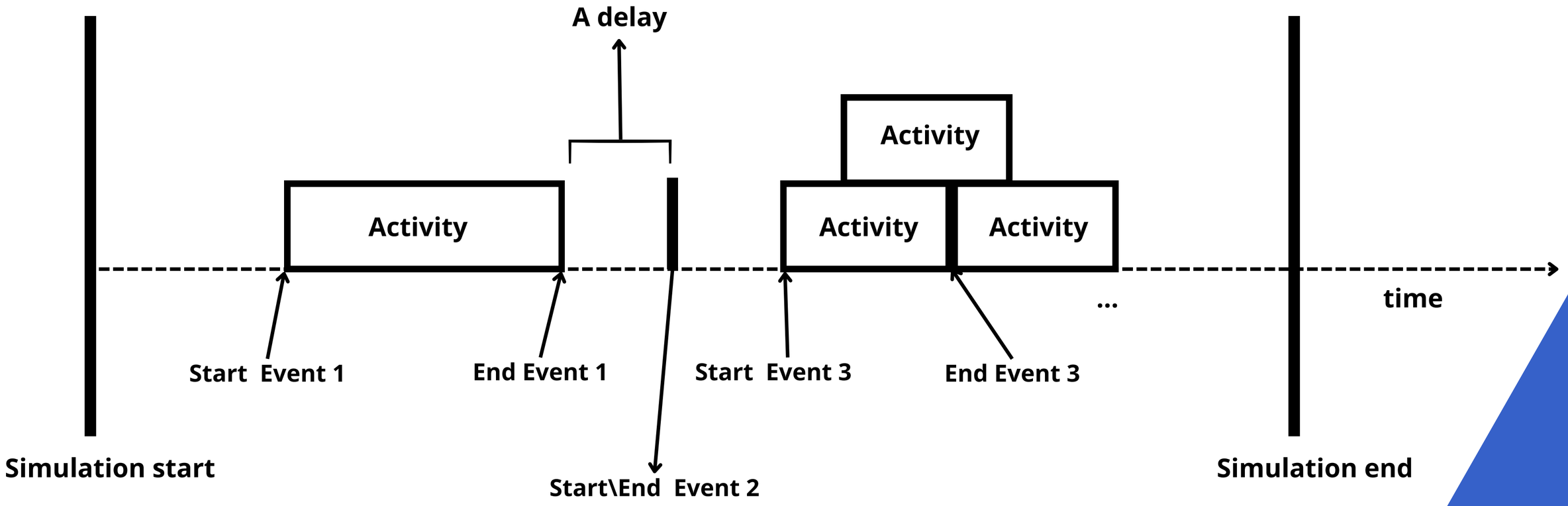
Discrete-Event Simulation

DES forms:

- **Incremental time progression: time is broken up into small time slices and the system state is updated according to the set of events/activities happening in the time slice.**
- **Next-event time progression : this represents the core advantage of DES, because simulation time directly jump to the occurrence time of the next event, our code above simply inspects the scheduled event times of all pending events and updates SimTime to the minimum among them.**

Both forms of DES contrast with continuous simulation.

Discrete-Event Simulation Schema



Some F.A.Q.

- **Can we model two events that take place at the same time?**
  - **Yes! in Discrete Event Simulation (DES), two or more events can occur at the same simulation time. However, since DES is sequential by nature, the simulator must still process them one at a time, in a specific execution order.**
- **Is randomness allowed in DES?**
  - **Yes! and it's one of the main strengths of DES. DES often uses randomness to model uncertainty in real-world systems, such as random arrival times, random service durations etc.**

2 Introduction to SimPy

Python Generator Functions

A generator is a special kind of function that lets you pause and resume execution. It uses the *yield* keyword instead of return. It is the novelty on which SimPy is based!

```
def count_up_to(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

```
>>> generator = count_up_to(4)
>>> print(next(generator))
0
>>> print(next(generator))
1
>>> print(next(generator))
2
>>> print(next(generator))
3
```

The body of the generator function is executed by a reiterated call to the generator's next() method until an exception is raised. When the yield statement is executed, the state of the generator is frozen and the value of the expression list is returned to the next() call.

Why do we care?

In SimPy, each process (e.g., a bus, a customer) is modeled as a generator function.
This allows processes to wait (*yield*) for events like delays, resources (explained later), or other conditions and then resume exactly where they left off.

overview

- SimPy (Simulation in Python) is a next-event time simulation framework based on standard Python.
- Its event dispatcher is based on Python’s generators so it enables users to model active components such as customers, vehicles, or agents as simple Python generator functions.
- SimPy is released as open source software and, since it is implemented in pure Python and has no dependencies, SimPy runs on Python 3.

components

ENVIRONMENT

- A simulation environment manages the simulation time as well as the scheduling and processing of events. It also provides means to step through or execute the simulation.

- Normal simulations use Environment (as fast as possible). For real-time simulations, SimPy provides a RealtimeEnvironment (wall-clock time) where we can manually step through the simulation event by event (step()).

- We can run our simulation until there are no more events, until a certain simulation time is reached, or until a certain event is triggered (until parameter).

components

PROCESSES

The main actors in the simulation. A process in SimPy is a Python generator function that describes the behavior of an entity over time (e.g., a machine, a customer, a bus, etc.).

Processes interact between them in 3 ways:

- **Sleep until woken up (passivate/reactivate)**
- **Waiting for another process to terminate**
- **Interrupting another process**

components

EVENTS

If processes are the main actors, Events can be seen as the actions.
An Event represents something that may happen in the future, and processes can wait for it. Everything in SimPy is built around events (e.g. timeouts, resource requests, custom signals..). An Event can be in 3 states:

- Might happen (not triggered)
- Is going to happen (triggered)
- Has happened (processed).

Initially, events are not triggered and just objects in memory.
If an event gets triggered, it is scheduled at a given time and inserted into SimPy's event queue. An event becomes processed when SimPy pops it from the event queue and calls all of its callbacks.

SimPy components

RESOURCES

The resource itself is some kind of a container with a, usually limited, capacity. Processes can either try to put something into the resource or try to get something out.
If the resource is full or empty, they have to queue up and wait.
Different types of resources exist.
Can also decide a priority access (Priority/PreemptiveResource).

A simple example

Let's try to simulate a simple scenario where multiple workers (processes) try to use a shared resource.

```python
def worker(env, name, machine):
    print(f'{name} arrives at the machine at time {env.now}')

    # Request the shared resource
    with machine.request() as req:
        yield req  # Wait until the resource is available
        print(f'{name} starts using the machine at time {env.now}')

        # Simulate some processing time
        processing_time = 3
        yield env.timeout(processing_time)  # This is an event!

        print(f'{name} finished at time {env.now}')
```

A note: simpy environment recognize a process only if it contains in his body at least a yield call.

A simple example

After we defined the worker, we can already start setting up the environment and start the simulation!

```python
# Setup the environment
env = simpy.Environment()

# Create a shared resource with capacity 1
machine = simpy.Resource(env, capacity=1)

# Create processes (they compete for the machine)
env.process(worker(env, 'Worker A', machine))
env.process(worker(env, 'Worker B', machine))
env.process(worker(env, 'Worker C', machine))

# Run the simulation
env.run()
```

```
Simulation starts
Worker A arrives at the machine at time 0
Worker B arrives at the machine at time 0
Worker C arrives at the machine at time 0
Worker A starts using the machine at time 0
Worker A finished at time 3
Worker B starts using the machine at time 3
Worker B finished at time 6
Worker C starts using the machine at time 6
Worker C finished at time 9
Simulation ends
```

A simple example, extended

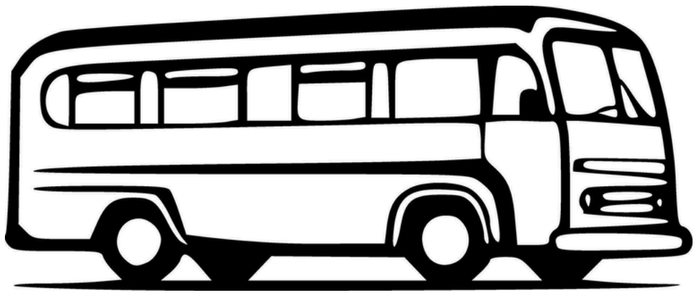
We can also extend the example by introducing an interruption, which, as mentioned before, is a way to make processes interact.

```python
def worker(env, name, machine):
    print(f'{name} arrives at the machine at time {env.now}')
    try:
        with machine.request() as req:
            yield req
            print(f'{name} starts using the machine at time {env.now}')

            # Simulate some processing time
            processing_time = 10
            yield env.timeout(processing_time)

            print(f'{name} finished at time {env.now}')
    except simpy.Interrupt as i:
        print(f'{name} was interrupted at time {env.now} (reason: {i.cause})')

def interrupter(env, target):
    yield env.timeout(4)  # Wait for a bit
    print(f'Interrupter fires at time {env.now}')
    target.interrupt('Intrerruption caused by interrupter')
```

```
Simulation starts
Worker A arrives at the machine at time 0
Worker B arrives at the machine at time 0
Worker C arrives at the machine at time 0
Worker A starts using the machine at time 0
Interrupter fires at time 4
Worker A was interrupted at time 4 (reason: Intrerruption caused by interrupter)
Worker B starts using the machine at time 4
Worker B finished at time 14
Worker C starts using the machine at time 14
Worker C finished at time 24
Simulation ends
```

3 Bus allocation scenario

Scenario



We are organizing an event, and for logistical reasons participants will need to park far from the event location. To make things easier, we are offering a shuttle bus that will take them directly to the event.

How many buses should we allocate to balance average waiting time and costs?

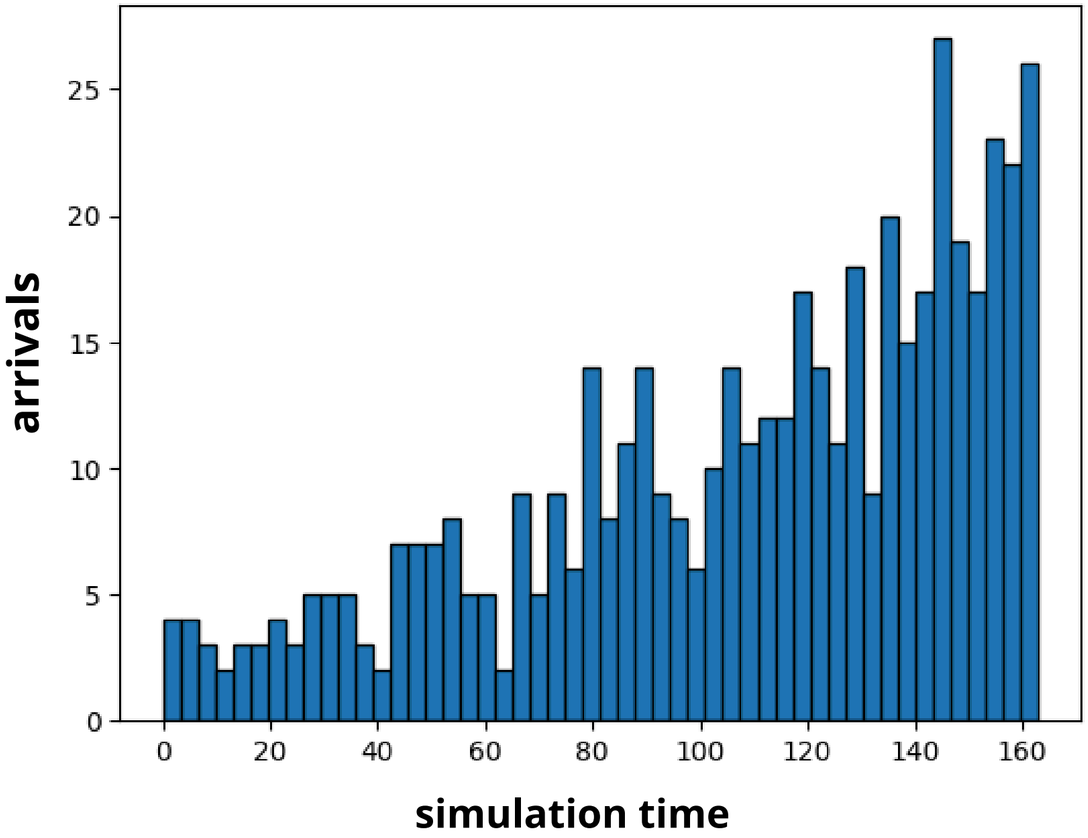
Problem Setup: arrivals

People arrives at the parking following an exponential distribution, to model the fact that usually there are peak hours where the frequency of arrivals increases, i modeled the problem in this way:

```
if now < PEAK_HOUR:
    new_lambda = BASE_LAMBDA + PEAK_FACTOR * (now / PEAK_HOUR)**2
else:
    new_lambda = BASE_LAMBDA + PEAK_FACTOR
yield env.timeout(random.expovariate(new_lambda))
```

Lambda Regulates the average arrival rate of passengers: 1/lambda is the average time between arrivals. So if lambda grows, the average time between arrivals decreases.

Problem Setup: arrivals



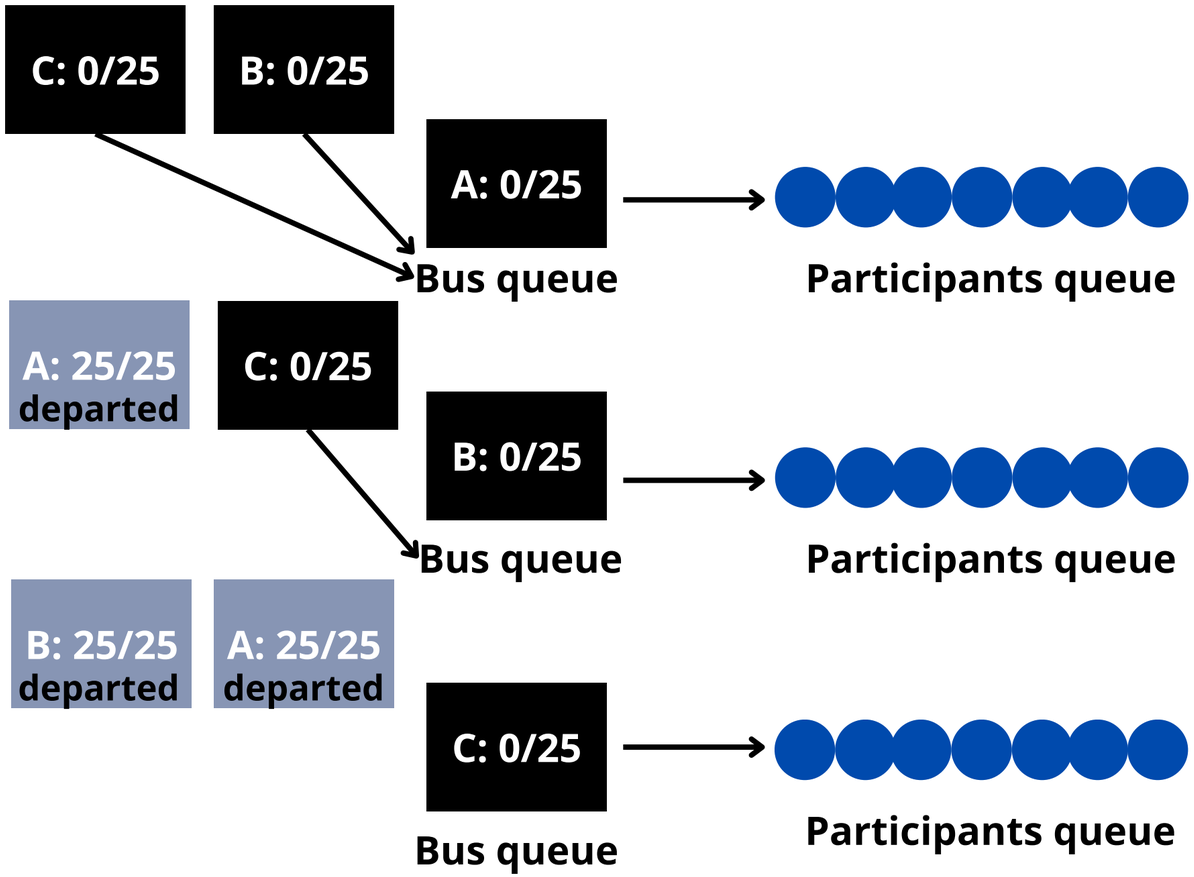
PEAK_HOUR: 150

Problem Setup: buses

Buses are the other main actor of this simulation:

- **Each bus is assumed to have a capacity of 25, seats are not modeled as a resource (it is easier to treat the participants as a resource), so seats are just a counter inside the bus process.**
- **A bus must be full before participants are allowed to board another bus waiting at the park.**
- **Once a bus is full, it departs, and the total travel time is a randomly generated from gaussian with mean 15 and standard deviation 2 . If another bus is available, participants begin boarding it.**

Queue mechanism



- The bus queue is implemented as a resource (a store) with capacity 1 (Only a bus at a time is available).
- Since the store resource is a FIFO, also the participants queue is implemented as a store, this time with unlimited capacity. Once again: the resources are the participants.

Simulation

First simulation aimed to find a reasonable amount of buses to allocate.
Some reminders:

- **Event participants: 500**
- **Buses Capacity: 25**

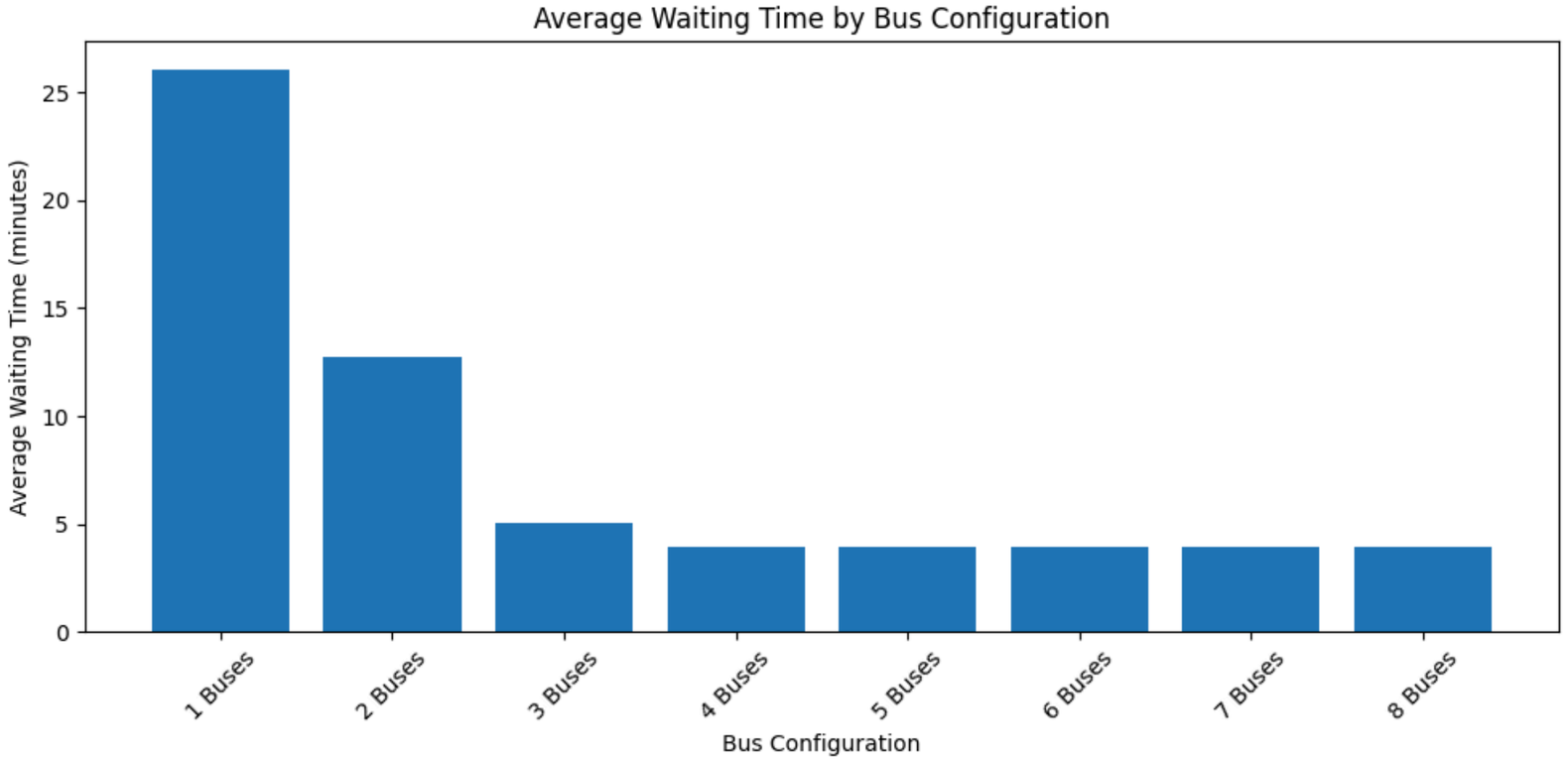
The simulation has been tested with various numbers of buses:

bus_amount = [1,2,3,4,5,6,7,8]

and each configuration was simulated 500 times to obtain more robust statistics.

```
17.50: Participant 18 arrives and joins general queue.
17.50: Bus_1: boarded passenger 18. Total: 18/25.
17.61: Participant 19 arrives and joins general queue.
17.61: Bus_1: boarded passenger 19. Total: 19/25.
17.86: Participant 20 arrives and joins general queue.
17.86: Bus_1: boarded passenger 20. Total: 20/25.
19.38: Participant 21 arrives and joins general queue.
19.38: Bus_1: boarded passenger 21. Total: 21/25.
19.94: Participant 22 arrives and joins general queue.
19.94: Bus_1: boarded passenger 22. Total: 22/25.
20.53: Participant 23 arrives and joins general queue.
20.53: Bus_1: boarded passenger 23. Total: 23/25.
20.59: Participant 24 arrives and joins general queue.
20.59: Bus_1: boarded passenger 24. Total: 24/25.
21.61: Participant 25 arrives and joins general queue.
21.61: Bus_1: boarded passenger 25. Total: 25/25.
21.61: Bus_1 is full (25 passengers). now departing!

21.61: Bus_2 is at the stop and active for boarding.
21.93: Participant 26 arrives and joins general queue.
21.93: Bus_2: boarded passenger 26. Total: 1/25.
22.04: Participant 27 arrives and joins general queue.
22.04: Bus_2: boarded passenger 27. Total: 2/25.
```

Simulation



3 buses start to be a reasonable amount for what we can see, with an average of 4.94 minutes.

Starting from 4 buses we can see there is basically no improvement:

- **4 buses: 3.95**
- **5 buses: 3.93**
- **6 buses: 3.93**
- **...**

Simulation: Smart Driver

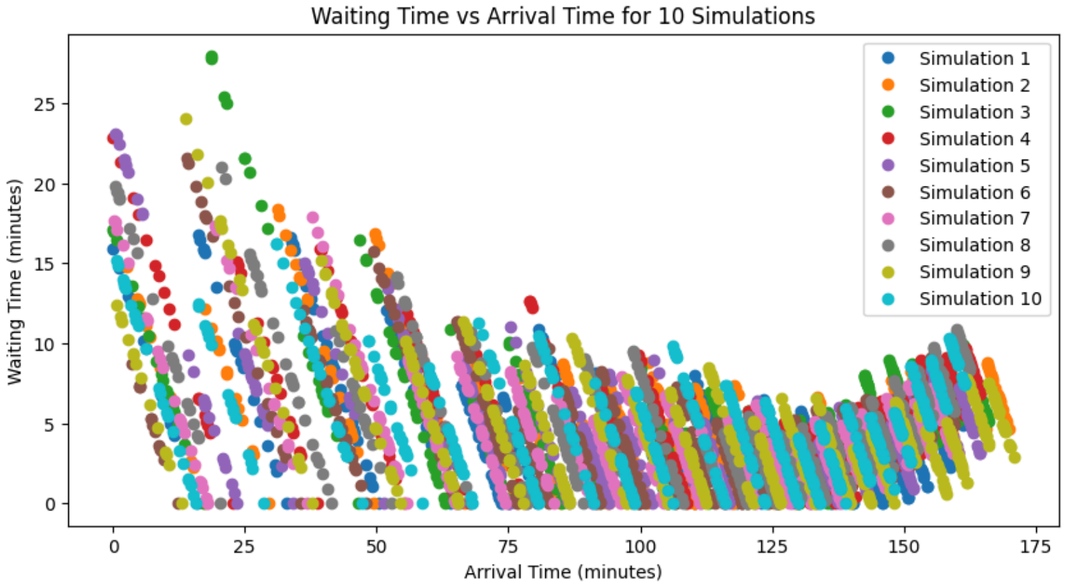
I wanted to check the longest time a participant had to wait for each simulation and i noticed that even for cases where we allocated >3 buses the maximum waiting time could reach up to 35 minutes.
Reason: the rule that states “a bus must be full before departing” can produce large waiting time in low frequency arrivals moments.

I wanted to implement a Smart Driver system where the driver decides to depart even if the full capacity has not been reached:

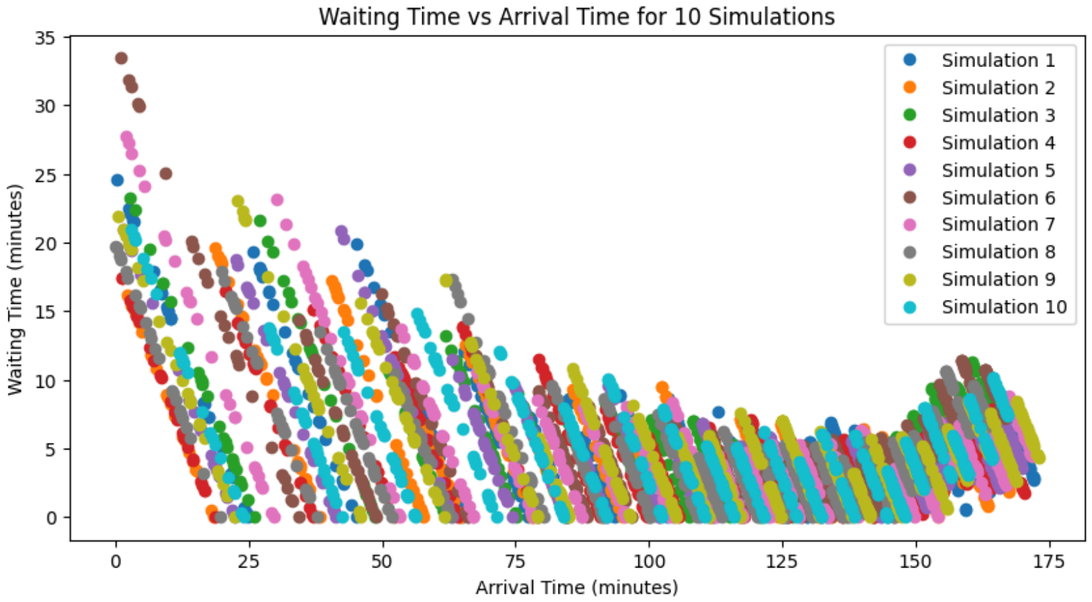
- **A bus can depart if there are at least X people on board.**
- **The last passenger boarded at least Y minutes ago.**
- **At least Z minutes have passed.**

Smart Driver: AND Conditions

By putting all conditions in AND we obtain nearly the same behavior as the No-Smart Driver scenario, but reduced maximum waiting time in some simulations.
Average waiting time of 4.55 vs 4.94.
Different amount of buses obtained ~0.15 decrement.



Smart time = 10
Smart Percentage = 0.5
Last passenger boarding time = 2



No Smart Driver

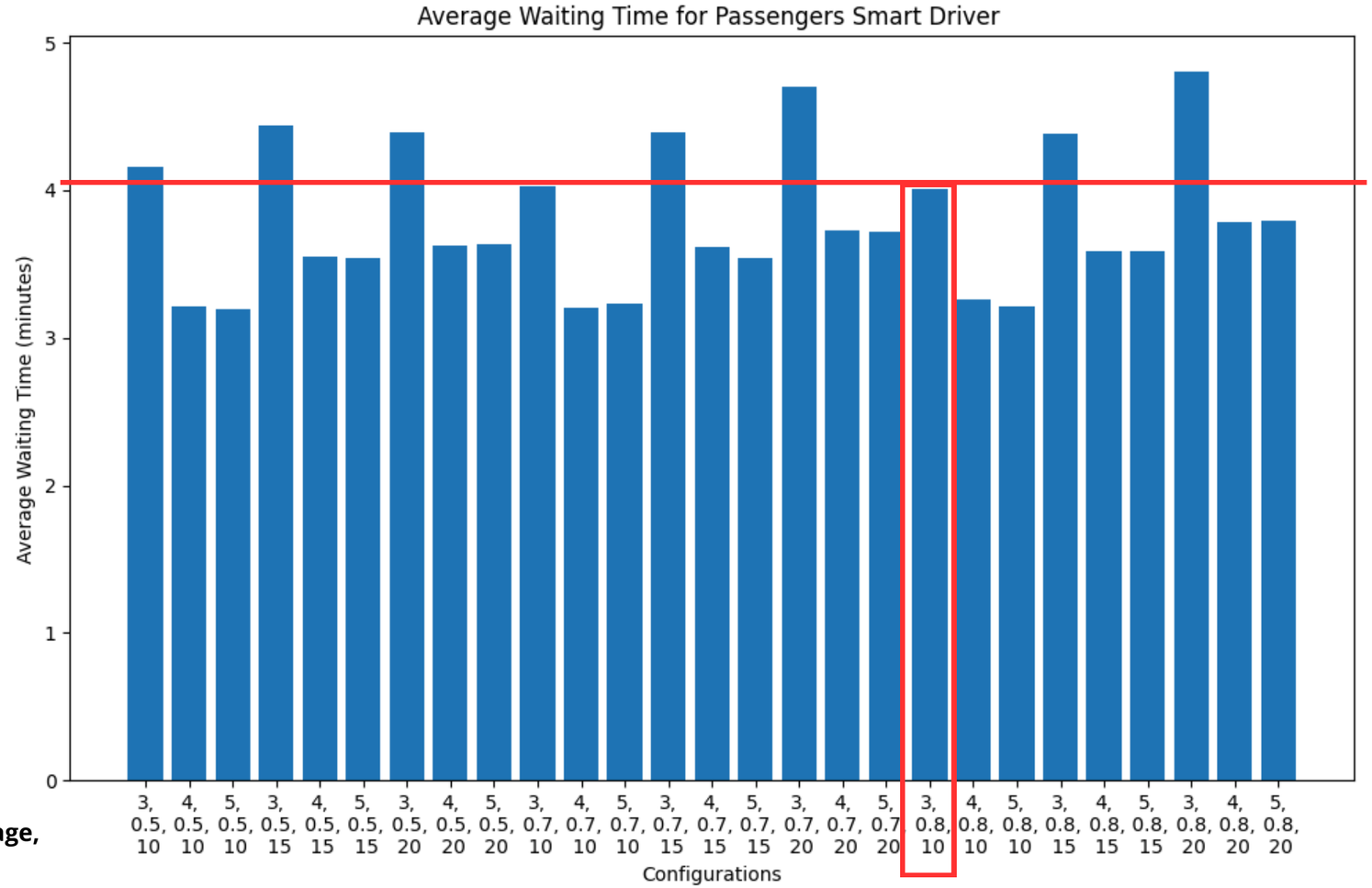
Smart Driver: 2 out of 3 Conditions

I tried to make the conditions less restrictive, ensuring that 2 out of 3 were enough for the bus to depart.

If we want to keep 3 Buses, in the best case we can achieve an average waiting time of 3.97 for these conditions! Basically as good as having 4 or more buses in the No smart driver scenario!

Another improvement in this case is that the average waiting time for 4 buses went rom 3.95 to 3.21.

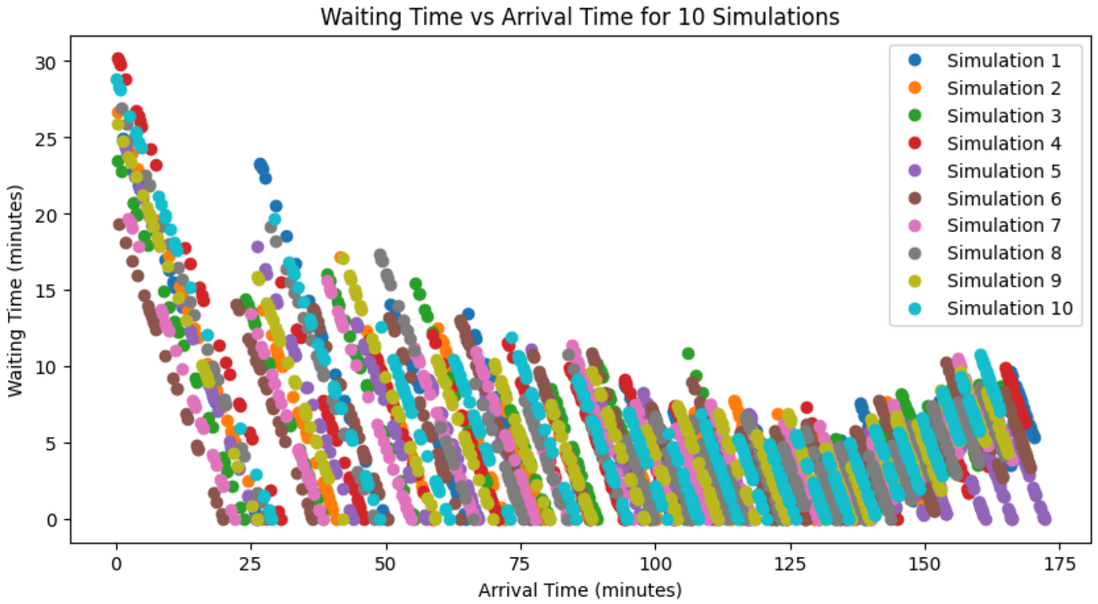
Smart Driver: OR Conditions



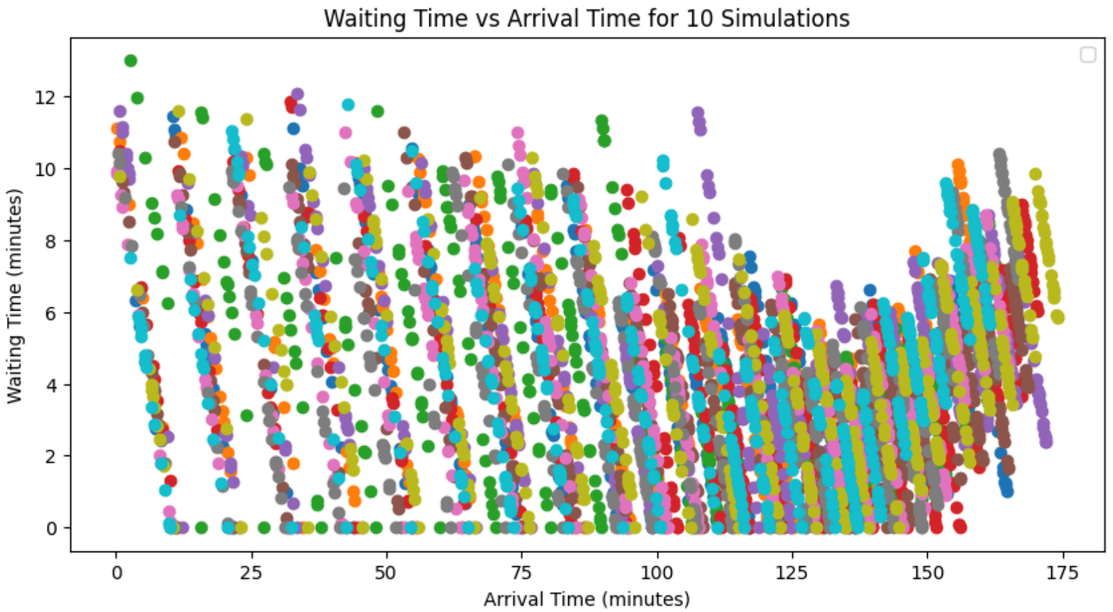
n_navette, smart_percentage, smart_time

“Last passenger boarding time” parameter was fixed to 2 in this simulation

No Smart Driver vs Smart Driver (OR)



No Smart Driver



Smart Driver (OR)

More buses ≠ shorter waiting times

- More buses do not automatically mean shorter waiting times.
- Smart management is key (e.g., flexible departures, not always full).
- More buses may reduce waiting times but also increase costs.

Appendix: Classes

```python
# Bus class representing a bus in the simulation
class Bus:
    def __init__(self, env, name, capacity):
        self.env = env
        self.name = name
        self.capacity = capacity # Maximum number of passengers this bus can carry
        self.passengers_on_board = 0 # Counter for passengers currently on board
        self.traveling = False # Indicates if the bus is currently traveling

# Participant class representing a client in the simulation
class Participant:
    def __init__(self, env, name, time_arrival):
        self.env = env
        self.name = name
        self.time_arrival = time_arrival # Time when the client arrived at the bus stop
        self.time_boarded = None  # Time when the client boarded the bus
        self.time_departure = None  # Time when the client finished their journey (simplified)

# Used to store the configuration of the simulation
class Configuration:
    def __init__(self, num_navette, smart_driver, smart_time, smart_percentage, max_passengers):
        self.num_navette = num_navette
        self.smart_driver = smart_driver # Indicates if the bus driver is smart
        self.smart_time = smart_time # Time the bus can wait before deciding to leave
        self.smart_percentage = smart_percentage # Percentage of bus capacity that must be filled before the bus departs
        self.max_passengers = max_passengers  # Maximum number of passengers for this configuration
        self.history = []  # List to keep track of passengers and their boarding and disembarking times
```

Appendix: participants and bus process

```python
# Function to generate participants (clients) in the simulation
def participant_generator(env, passenger_queue, history, tempo_massimo_simulazione, MAX_PASSENGERS, PEAK_HOUR, BASE_LAMBDA, PEAK_FACTOR):
    i = 0
    while len(history) < MAX_PASSENGERS:  # Limit the number of clients to MAX_PASSENGERS
        now = int(env.now)
        if now < PEAK_HOUR:
            new_lambda = BASE_LAMBDA + PEAK_FACTOR * (now / PEAK_HOUR)**2
        else:
            new_lambda = BASE_LAMBDA + PEAK_FACTOR

        # Generate a new participant based on the adjusted arrival rate
        yield env.timeout(random.expovariate(new_lambda))
        i += 1
        print(f'{env.now:.2f}: Participant {i} arrives and joins general queue.')

        # Participants join a general queue, not looking for a specific bus right away
        passenger = Participant(env, i, env.now)
        history.append(passenger)  # Add the participant to history for final report

        passenger_queue.put(passenger)
        if env.now > tempo_massimo_simulazione:
            break
```

```python
def bus_process(env, bus_obj, bus_attivi_per_imbarco, passenger_queue, SMART_DRIVER, SMART_TIME, SMART_PERCENTAGE, TRAVEL_TIME):
    # Bus cycle
    while True:
        # The bus tries to put itself in the active boarding queue
        yield bus_attivi_per_imbarco.put(bus_obj)

        # After being put in the queue, the bus can board passengers
        print(f'\n{env.now:.2f}: {bus_obj.name} is at the stop and active for boarding.')

        bus_obj.passengers_on_board = 0 # Counter for passengers on board
        current_passengers = [] # A list to keep track of current passengers boarding this bus
        time_idle = env.now  # Time when the bus started waiting for passengers
        time_from_last_boarding = 0  # Time since the last boarding
        time_of_last_boarding = env.now # Time of the last boarding action

        # The bus will board passengers until it is full or the SMART_DRIVER conditions are met
        while bus_obj.passengers_on_board < bus_obj.capacity and (not SMART_DRIVER or (env.now - time_idle <= SMART_TIME
                                                                                       and bus_obj.passengers_on_board <= bus_obj.capacity * SMART_PERCENTAGE
                                                                                       and time_from_last_boarding < 2)):
            # Wait for a passenger to be available in the queue
            passenger = yield passenger_queue.get()
            current_passengers.append(passenger)

            # update the frequency of boarding
            if SMART_DRIVER:
                time_from_last_boarding = env.now - time_of_last_boarding
                time_of_last_boarding = env.now

            # Once a passenger is available, board them on THIS bus
            env.process(client(env, passenger, bus_obj))
            passenger.time_boarded = env.now  # Record the time the passenger boarded the bus
            bus_obj.passengers_on_board += 1

            print(f'{env.now:.2f}: {bus_obj.name}: boarded passenger {passenger.name}. Total: {bus_obj.passengers_on_board}/{bus_obj.capacity}.')

        print(f'{env.now:.2f}: {bus_obj.name} is full ({bus_obj.passengers_on_board} passengers). now departing!')

        for passenger in current_passengers:
            passenger.time_departure = env.now

        # Once full, the bus "removes itself" from the active boarding bus store
        # This allows the next bus in line to activate.
        yield bus_attivi_per_imbarco.get() # The bus removes itself from the "active for boarding" state

        # Simulate the journey
        bus_obj.traveling = True
        yield env.timeout(TRAVEL_TIME)
        bus_obj.traveling = False

        print(f'{env.now:.2f}: {bus_obj.name} has arrived at destination and is unloading passengers. Now returning empty.')
        bus_obj.passengers_on_board = 0 # Unload passengers
        # The bus is now available for the next cycle (will return to boarding queue)
```

Appendix: simulation cycle example

```
# Simulate for different configurations
for TOTAL_BUSES in bus_amount :
    for SMART_DRIVER in smart_driver:
                # Create a new configuration for each combination of parameters
                configuration = Configuration(TOTAL_BUSES, SMART_DRIVER, SMART_TIME, SMART_PERCENTAGE, MAX_PASSENGERS)
                for i in range(100):  # Run 100 simulations for each configuration

                    env = simpy.Environment()

                    # This is the queue where passengers wait when they arrive
                    passenger_queue = simpy.Store(env)

                    # This Store manages which bus is ACTIVE for boarding at any given moment
                    # It has a capacity of 1, so only ONE bus can be placed here at a time.
                    # Buses "compete" to be here.
                    active_buses = simpy.Store(env, capacity=1)

                    # Start the bus processes
                    for i in range(TOTAL_BUSES):
                        navetta_obj = Bus(env, f'Bus_{i+1}', BUS_CAPACITY)
                        env.process(bus_process(env, navetta_obj, active_buses, passenger_queue,
                                                SMART_DRIVER= SMART_DRIVER,
                                                    SMART_TIME=SMART_TIME,
                                                        SMART_PERCENTAGE=SMART_PERCENTAGE,
                                                            TRAVEL_TIME=TRAVEL_TIME))

                    # Start the passenger generator
                    history = []  # List to keep track of passengers for the final report
                    env.process(participant_generator(env, passenger_queue, history,
                                                      tempo_massimo_simulazione=SIMULATION_TIME,
                                                          MAX_PASSENGERS=MAX_PASSENGERS,
                                                              PEAK_HOUR=PEAK_HOUR,
                                                                  BASE_LAMBDA=BASE_LAMBDA,
                                                                      PEAK_FACTOR=PEAK_FACTOR))

                    # Run the simulation
                    print(f'--- Simulation started for {SIMULATION_TIME} minutes ---')
                    env.run(until=SIMULATION_TIME)
                    print('--- Simulation end ---')

                    configuration.history.append(history)  # Save the passenger history in the configuration
                configurations.append(configuration)
```

Sources

- https://heather.cs.ucdavis.edu/matloff/public_html/156/PLN/DESimIntro.pdf#page=8&zoom=100,117,857
- https://www.youtube.com/watch?v=Bk91DoAEcjY (SimPy Talk)
- https://wiki.python.org/moin/Generators
- https://simpy.readthedocs.io/en/latest/index.html
- https://simpy.readthedocs.io/en/latest/topical_guides/events.html
- https://simpy.readthedocs.io/en/latest/topical_guides/process_interaction.html
- https://simpy.readthedocs.io/en/latest/topical_guides/events.html
- https://simpy.readthedocs.io/en/latest/topical_guides/resources.html

code:

https://github.com/fefex302/Discrete-Event-Simulation-for-Bus-Allocation