Shepherd

Release 0.1

Pliche Developer

Jul 17, 2023

CONTENTS

Shepherd is a web application using the Django Python web development framework allowing non-experts in Reinforcement Learning (RL) to easily use state-of-the-art RL techniques. It acts as a bridge between web clients, that connect to it over the network (using JSON commands sent over HTTP), and Reinforcement Learning agents available in the Stable-Baselines3. No specific dependencies must be installed on the client's side, nor must a specific programming language be used. Shepherd presents itself to the RL algorithms as a fully standard OpenAIGym environment. At the core of Shepherd, a Transfer Learning method is used to allow RL agents (all under one Shepherd agent), each trained by a different client of Shepherd, to advise and help improve each other. This is why Shepherd is comparable to a single agent, multiple executions setting, without the need to fundamentally modify the RL algorithms from Stable-Baselines3.

We describe below the components of Shepherd, namely: i) the database allowing to keep track of users/clients, and of their RL agents; and ii) the communication workflow between client and server, letting a client's environment to sporadically send observations to an RL agent on the server side, and getting actions in return.

In the following sections, by the term "user", we mean a person that possesses an environment applicable to the use of an RL algorithm. For a user environment to be able to benefit from our service, the only requirements are that observations can be extracted from the environment, and that the environment can take actions as input. The only contribution left from the user is to implement a client application, establishing the connection with our server, sending observations and receiving actions. However, this part is relatively trivial to implement, and we provide an example.

Note: To use Shepherd, you do not need to know how RL works, however, you do need to have a problem applicable for Reinforcement Learning.

CHAPTER

ONE

USER GUIDE

1.1 Shepherd agent

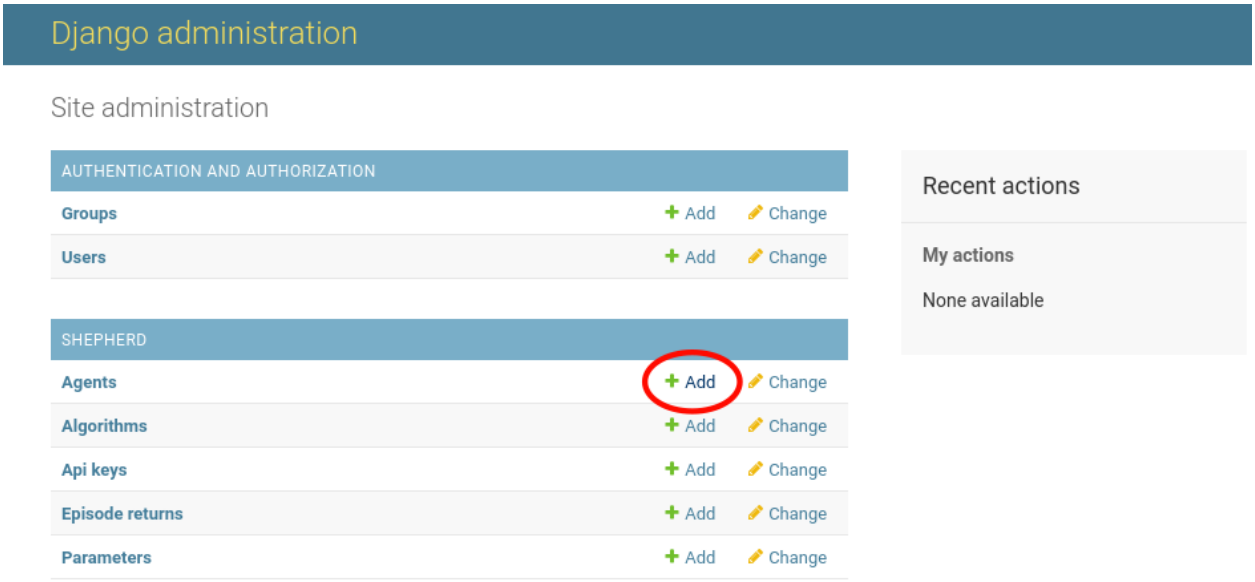
New users are added to the database by an internal staff member through the admin page. The onfly information required is a name to identify the user; we use API keys to log users in instead of a user password. A User can possess more than one Shepherd Agent (one per environment that they have, or one per different algorithm on one environment to compare algorithms, for instance).

Once a Shepherd user exists, a Shepherd agent associated to that user can be created. To avoid the need for a user name and password, we use generated **API keys** by the UUID module. The particularity of our usage of API keys is that one API key is also associated to one Shepherd Agent. When logging in to Shepherd, the client sends an API key to the server, which retrieves the corresponding API key object, linked to the user's corresponding Agent. In the case where a user has multiple Shepherd agents, they would also have multiple API keys, one per agent. The advantage of this setting is that if one API key is leaked, only one of the user's Shepherd Agents is leaked.

The specification of the action and observation spaces of the user's environment are required for the initialization of a Shepherd Agent. It is also at the agent creation time that an RL algorithm to be executed by the agent is chosen. New Shepherd agents are created through the admin page, as well as the parameter values to configure the algorithm run by an agent.

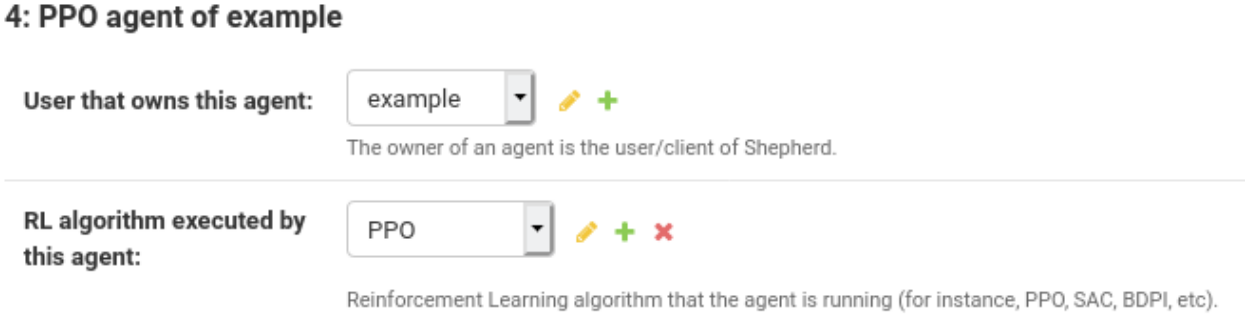
The difference between a Shepherd agent and a standard RL agent is that several standard RL agents can "run" one Shepherd agent: a Shepherd agent stores a configuration, specifying which algorithm must be executed, the parameters, as well as the latest generated model and results obtained.

1.1.1 Creating a Shepherd agent



The user owning the Shepherd agent must be specified. Only they (the user) can launch that specific Shepherd agent, access its learned models, see its learning curves, perform special actions such as deleting the agent and its models, etc.

The algorithm run by all RL agent instances under that Shepherd agent must also be specified. There exist several RL algorithms, and some might work better than others depending on the problem/environment (e.g., not all RL algorithms are compatible with continuous actions). The algorithm chosen during the creation of a Shepherd agent cannot be changed afterwards.



1.1.2 Action and observation spaces

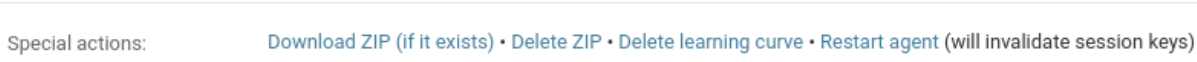
A Shepherd is always associated with one environment to be solved. This environment, run on the client's side, must send observations to the Shepherd agent, and receives actions in return.

Let's consider CartPole-v0, from the Gym. Its observations are in the form of vectors of four floats; each float is equal to 3.4028234663852886e+38 at its highest, and -3.4028234663852886e+38 at its lowest.

```
>>> import gym
>>> env = gym.make("CartPole-v0")
>>> obs = env.observation_space
>>> obs
Box(-3.4028234663852886e+38, 3.4028234663852886e+38, (4,), float32)
>>> actions = env.action_space.n
>>> actions
2
```

In the specs of the Shepherd agent on the admin site, the action and observation spaces must be expressed in valid JSON. We set the action space to two (corresponding to the two discrete actions available in CartPole); the observation space can eb expressed as follows: [[space], low, high].

There are few "special actions" that the user can do through the admin page; they can download data from the agent, such as the episode rewards and the agent's model, and reboot its learning, which is especially useful when the environment has been modified and that previous results become irrelevant.



1.1.3 Algorithms

An **Algorithm** has a fairly brief definition in our database; it has a name, often the abbreviated version of the name of an RL algorithm, such as PPO or A3C and a Boolean flag indicating whether it is compatible with environments with continuous actions or not. An Algorithm also has several **Parameters** associated to it. Algorithm objects are preexisting the creation of Users and Agents; we pre-populate the database with RL algorithms from Stable Baselines 3, but adding new algorithms that are not from Stable Baselines 3 is trivial.

1.1.4 Parameter values

When a new Shepherd agent is created, an RL algorithm must be chosen to be run by this agent. If they want to, the user can configure the algorithm through setting the value of some parameters, via the addition of **ParameterValue** objects in the database. This comes in handy when the user wants to try parameter values different from predefined default ones. The value chosen is stored in value_int, value_float, value_bool (exclusive) or value_str, depending on the *type* of the corresponding **Parameter** object (see below); all three other value attributes are set to null. A ParameterValue is associated to one Shepherd Agent, and to one Parameter.

1.1.5 Parameters

Each RL algorithm in the database uses a multitude of **Parameters** (e.g., learning rate, batch size, gamma, etc), all associated to that one algorithm. The type of the value this parameter can take is defined by an Integer in [1, 2, 3, 4]; if *type* is equal to 1, the value of the parameter must be an Integer, if *type* = 2, the value must be a floating point, etc. Only one of the attributes value_int, value_float, value_bool and value_str contains a value different than null, depending on the *type* attribute. Similarly to Algorithms, Parameters are pre-existing ParameterValues; we populate the database with parameters used by most RL algorithms, and set their default values, before the addition of Users and their Shepherd Agents.

1.1.6 Episode Returns

Multiple **EpisodeReturns** are associated to one Shepherd Agent. Each EpisodeReturn object stores a float (the sum of all rewards collected during one episode), and a date time field. Episode returns are used to plot the Agent's learning curve, automatically displayed on the admin page.

1.2 Example of a Shepherd client

The example code below logs in our Shepherd server using an API key. Once successfully logged, the code starts communicating with the shepherd/env/ view. Each of these communications must contain four information: an observation *obs*, a *reward*, a *done* flag and a dictionary *info*. All four information come from the user's environment. The observation should contain whatever information the agent should see to be able to learn the task. For instance, if the agent must control the setting of a smart thermostat, the observation could be the current room temperature and the current target temperature of the thermostat. The reward is a floating-point value; it is used to transmit to the agent an idea of how good its last action(s) was(were). In the thermostat example, the user of the thermostat could push a green or red button to indicate whether he is happy with the current temperature of the room, and the green button could correspond to a 1.0 sent to the server as reward, the red button as a 0.0.

The done flag indicates the end of an episode, and the start of a new one. In the case of the smart thermostat, an episode could end every 24 hours. *info* can remain an empty dictionary, and is empty in most RL applications. When an episode ends and that a new one starts, the environment often executes some sort of preparation for the next episode. In our example below, the CartPole environment from Gym explicitly calls a function called *reset()*, but it does not need to

be called that way. Typically, a reset procedure puts the environment back in an initial state, as when putting all pieces back on the board when preparing to play a new game of chess.

For each *obs*, *reward*, *done*, *info* communication from the client to the server, the server responds with an action from the agent. An action can be an integer, a float or even a vector of several values, depending on the task to be solved. In the example of the thermostat, the agent's action could be a target temperature. The agent does not return an action when *done* is True, but returns None instead. If the action received from the server is None, then the last *done* received by the server was True, meaning that an episode just ended.

```
import os
import sys
import json
import requests

import gym

def send_json_to_website(d, path):
    """ Sends "d", a dictionary, to the website, and returns the response as a Python␣
↪dictionary
    """
    r = requests.post('http://localhost:8000/'+ path, json=d)
    return r.json()

# Make the environment on client side (here, gym environment)
env = gym.make('CartPole-v0')
obs = env.reset()

# First communication: login user
ok = send_json_to_website({'apikey': your_api_key}, 'shepherd/login_user/')

if 'ok' in ok:
    print("Login successful")
    session_key = ok['session_key']

    # Start sending observations, in exhange of actions with the agent on the server side
    action = send_json_to_website({'obs': obs.tolist(), 'reward': 0.0, 'done': False,
↪'info': {}, 'session_key': session_key}, 'shepherd/env/')

    while True:
        print("action ", action)
        obs, reward, done, info = env.step(action['action'])
        action = send_json_to_website({'obs': obs.tolist(), 'reward': reward, 'done':␣
↪done, 'info': {}, 'session_key': session_key}, 'shepherd/env/')

        # End of an episode, beginning of a new one
        if action['action'] is None:
            assert(done)
            obs = env.reset()
            action = send_json_to_website({'obs': obs.tolist(), 'reward': 0.0, 'done':␣
↪False, 'info': {}, 'session_key': session_key}, 'shepherd/env/')

    # When the client wants to stop, its last communication is to return None as␣
↪observation
    action = send_json_to_website({'obs': None}, 'shepherd/env/')
```

(continues on next page)

(continued from previous page)

```
else:
    print("ERROR: Could not login to server")
```

1.2.1 On the Shepherd server side

In a conventional reinforcement learning setting, the agent is the driving force of the interactions between the agent and the environment. In a typical RL workflow, the agent generates an action, then prompts the environment for an observation, in exchange of that action. In our Shepherd framework, on the other hand, it is the environment on the client side that prompts the agent on our server for an action, in return of an observation. This role reversal calls for a carefully though out solution on the server side for standard RL algorithms to smoothly adapt to that modified communication workflow.

To leverage their Shepherd agent running on our server, users must first log in by sending their API key. Once the Shepherd agent linked to the particular API key received by the server is retrieved, a thread is created to run an execution of that Shepherd agent; the initialization of a thread looks for the latest save of any previous execution of the agent, to pick up where the last one ended. If one such save is found (i.e., a zip file in which neural networks weights have been saved), the thread loads it, otherwise, the thread starts learning from scratch. In any case, each running thread regularly saves its network weights in a directory shared by all executions of a given Shepherd Agent. The thread initialization also takes care of instantiating the Shepherd Environment (called ShepherdEnv-v0), with which the learning execution is actually going to interact.

Several threads can run the same Shepherd agent at the same time, as in the case of a web application allowing multiple users to log using the same API key. Although each user technically trains its own instance of the RL algorithm specified by the Shepherd agent, since threads advise each other using a transfer learning method, and share their saves, it is as if all users train the same Shepherd agent.

Once the User and the Agent have been identified, the user's client can start sending observations from their environment. All messages sent from the client to the server must not only contain an observation, but also a reward and a Boolean **done** indicating whether the episode is over or not. This information is stored by the targeted learning execution (or thread) immediately after have been received on the server end. Each thread has two queues as attributes: an observation queue, storing the latest information received from the client, and an action queue, in which the learning execution puts its actions. When a conventional RL algorithm (such as the ones in Stable Baselines 3) running on the server prompts the environment for an observation, reward and done information, the thread prompts the ShepherdEnv-v0 environment with the action selected as input. This custom gym environment puts the action in the thread's action queue, dequeues the thread's observation queue, and returns the output of the observation queue. To standard RL algorithms, ShepherdEnv-v0 ``feels" just like a regular gym environment, although it does not simulate the user's environment; the environment actually being learned by the agent is running on the client side. ShepherdEnv-v0 merely manages the communication between the client's environment and the RL algorithm running on the server.

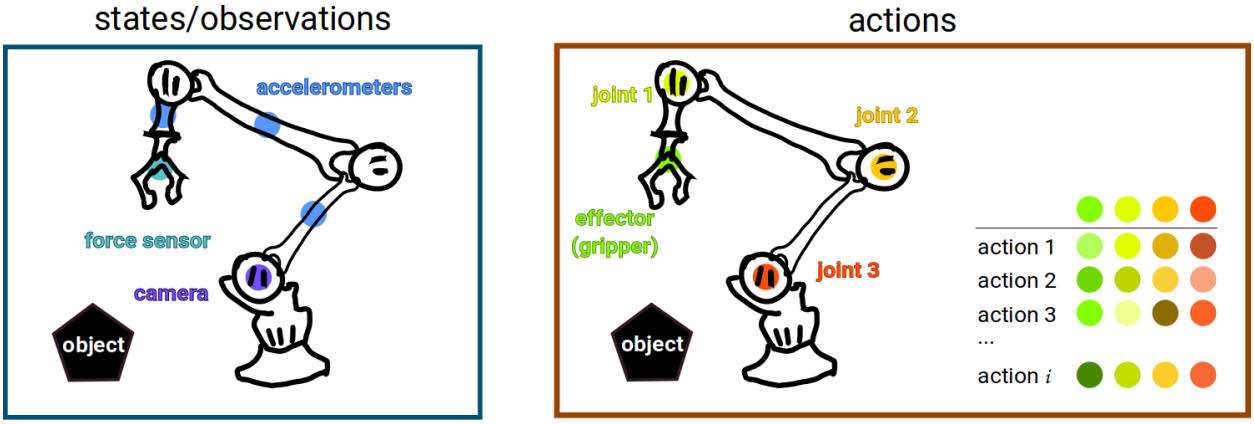
1.3 What is an Environment?

In our Shepherd agent creation section, and in our code example of a client sending observations to the server, we considered CartPole, a very simple simulated environment often used by RL researchers and AI students. In this section, we go a little further and imagine a real-life task to be solved, controlling a robotic arm, and detail how to abstract this task into an environment able to interact with a Shepherd agent. Note that an environment is in no way a simulator of the robotic arm; it can be viewed as a software bridge between the robot arm and the agent, allowing the agent to observe the current state of the arm, and to send actions to be executed by the arm. Reinforcement Learning can be applied to any real-life problem, as long as clear states and actions can be identified, as we describe below.

1.3.1 How to make an RL environment?

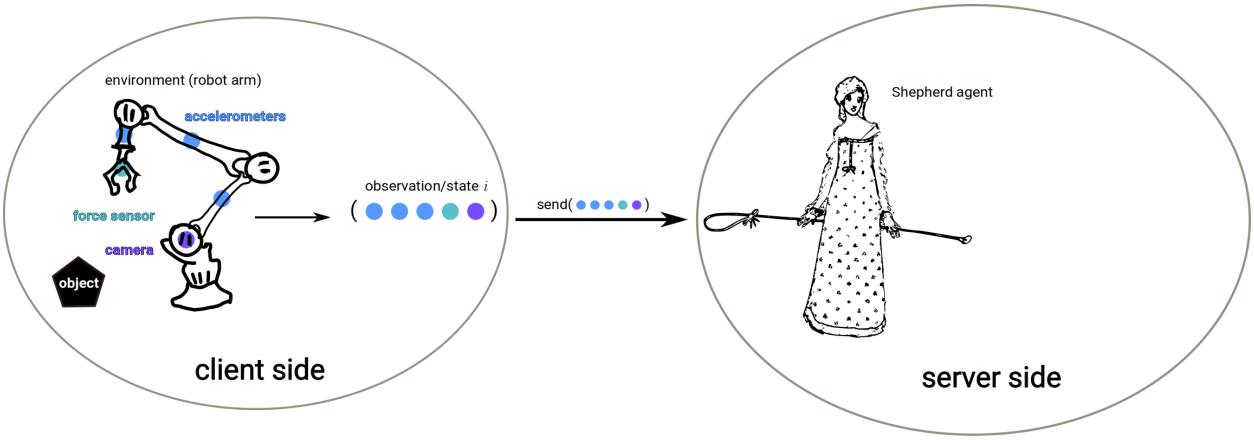
Let's say we have a robotic arm; we want this arm to learn how to manipulate an object using RL. Our robot arm has four sensors: three accelerometers, a camera at the base of the arm, and a force sensor on the effector, a gripper. At each timestep, the current state of the robot arm can be expressed as the concatenation of all 4 sensors readings, resulting in a vector of 4 floating point values, for instance. This vector can be sent to the reinforcement learning agent, which in turn will send an action based on the current state of the arm.

For the agent to choose an action be executed by the robot arm, there must be some actions available to the agent. Our arm has three joints, which can each be activated by a motor, and an effector (the gripper). Each motor controlling a joint can be activated given a floating point value, for instance, as well as the gripper. As a result, an action is expressed as a vector of four floats, each controlling one of the three joints and the gripper. In this particular example, since all four components of an action is a floating point value, there is an infinite amount of different actions at the disposal of the agent. Hence, the action space is said to be continuous, in opposition to discrete.

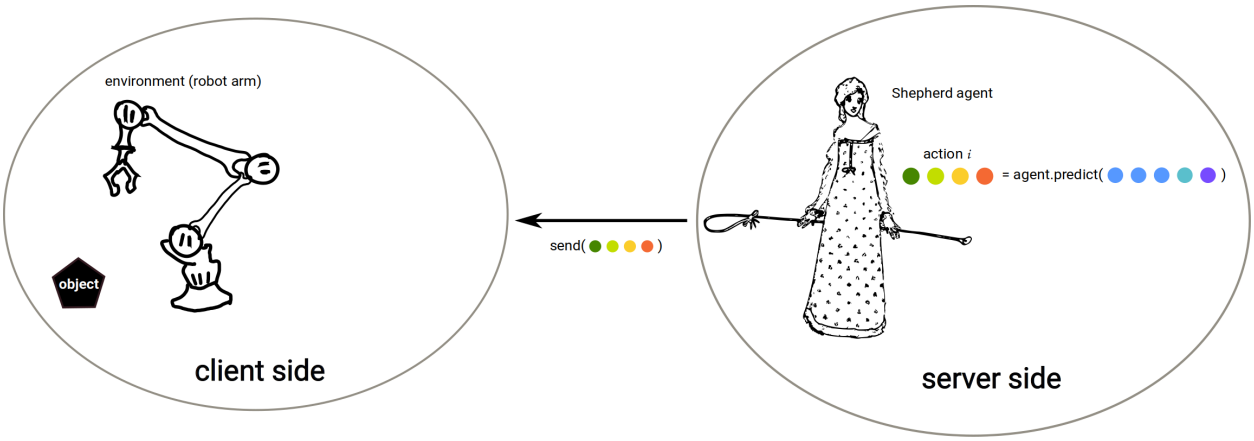


1.3.2 Interactions between environment and agent

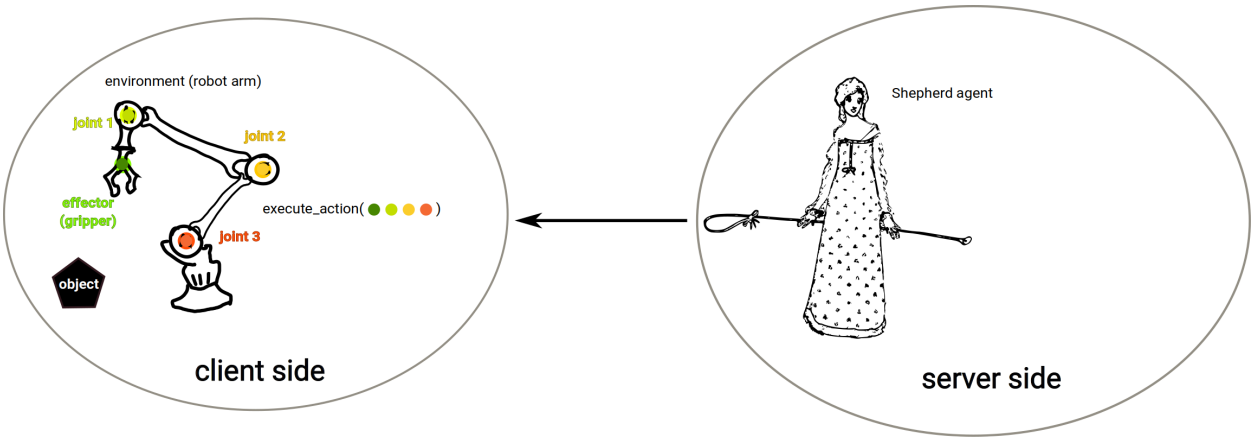
The first step is to have the environment on the client side feel gather an observation of the current state of the arm. The client sends this observation to the server, on which the Shepher RL agent is.



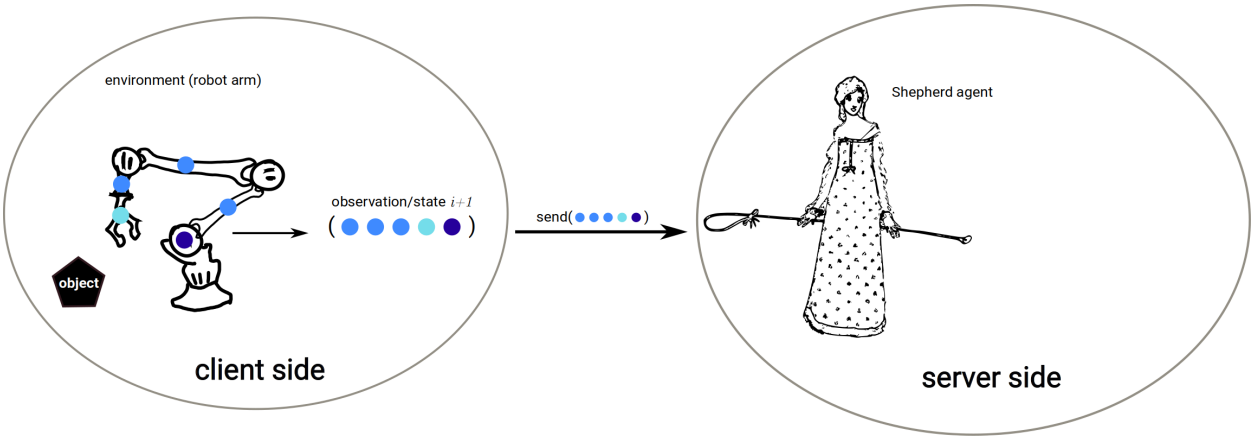
Based on the received observation, the Shepherd agent computes an action. The server sends the action to the client.



The client receives the action, passes it to the environment. The controller of the robot arm on the client side executes the action received from teh agent.

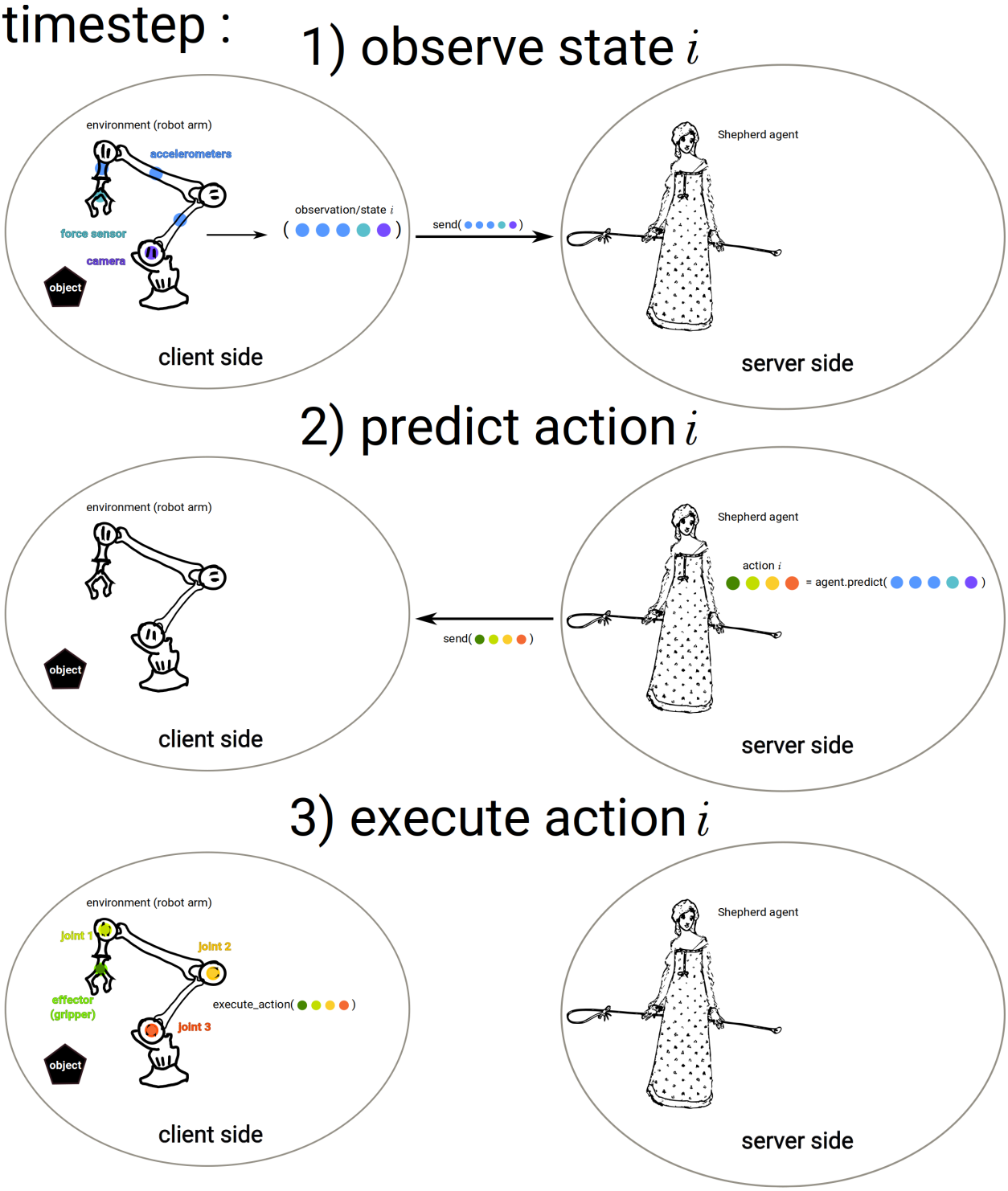


Afterwards, a new state can be observed. The cycle continues.



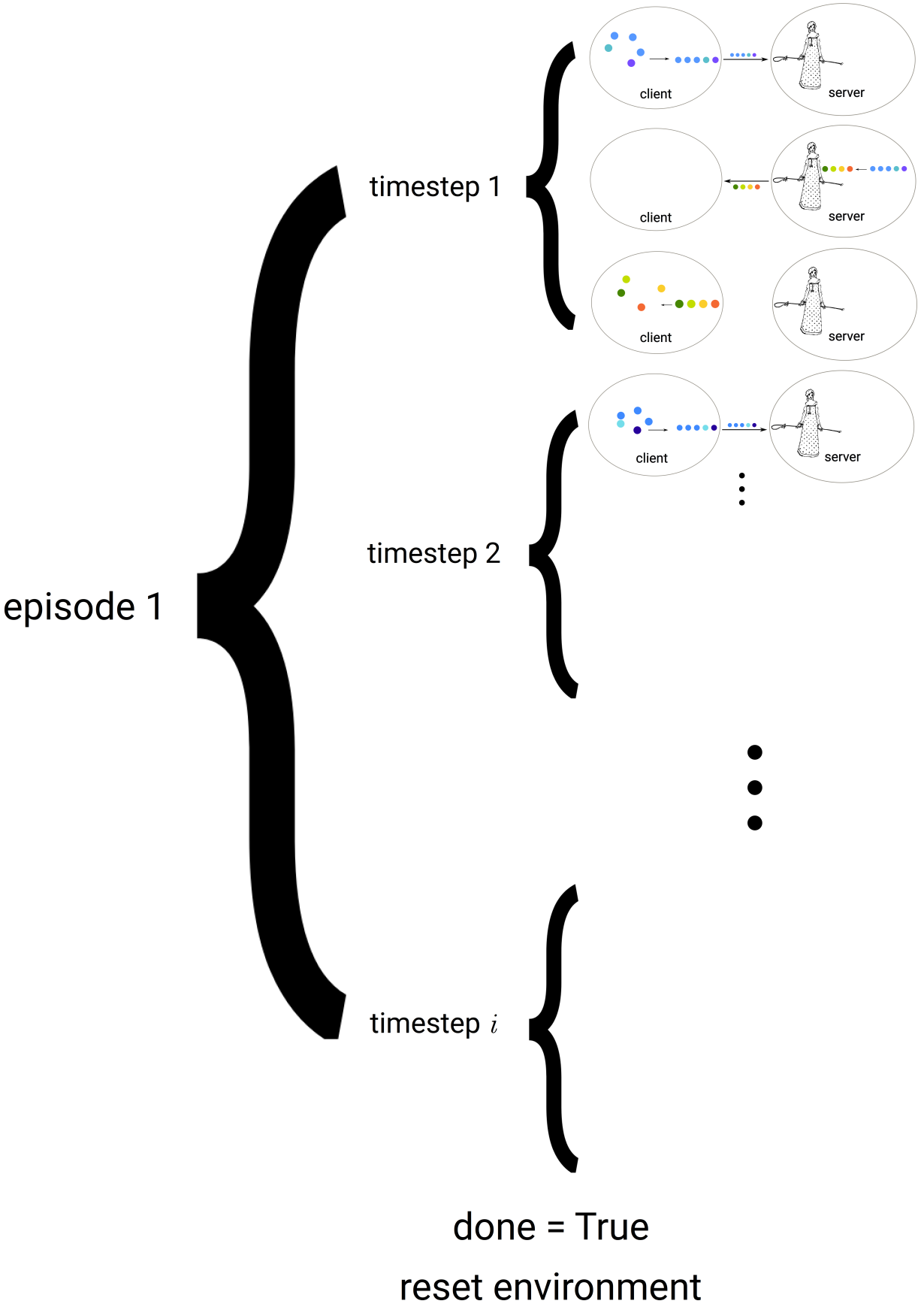
1.3.3 Timestep, episode and experience

A timesteps is composed of three steps mentioned above: 1) observing a state, 2) computing an action, 3) executing the action. Observing the new state resulting from executing the action is the first step of the next timestep.



An episode is composed of a number of timesteps. The number of timesteps in an episode can be determined by answering the following question: when can the task be considered "done"? In our robot arm example, we could say

that the task is over when the object has been picked up and put down on a target spot by the arm. A reinforcement learner is going to try to perform that task over and over again until it does it perfectly. The amount of timesteps in an episode could be however many timesteps occur before the object has correctly been placed on the target spot, hence, each episode has different length. In contrast, one could decide on an arbitrary, fixed amount of timesteps, after the episode ends, whether or not the goal has been reached.


client
server
client
server
client
server
client
server
timestep 1
timestep 2
timestep i
episode 1
done = True
reset environment

At the end of an episode, there must be a reset function, which puts the environment in an initial state. In our example, at the end of each episode, the object must be put back at its initial spot.

An experience is composed of several episodes.

1.3.4 One agent, several executions

Now let's say the user owns not one, but several robotic arms, and wants each of these arms to share their knowledge with each other to help them learn faster, instead of each learning the task from scratch on their own. Each of these arms can log in the Shepherd server with the same API key, corresponding to the one Shepherd agent executing the algorithm the user wants to use, and start sending its observation to that agent. This way, all arms send their observations and rewards to the same Shepherd agent, which in turn makes sure that an efficient transfer of knowledge between robot arms occurs. This has a two-fold advantage: all arms can potentially learn the task faster, and if the user wants to plug yet another robotic arm, this lastly added arm will start executing an already well-trained behavior, retrieved from the other arms' saved models.