Job-shop scheduling problem

Job-shop scheduling, **the job-shop problem** (**JSP**) or **job-shop scheduling problem** (**JSSP**) is an optimization problem in computer science and operations research. It is a variant of optimal job scheduling.

In a general job scheduling problem, we are given *n* jobs J_1, J_2, ..., J_n of varying processing times, which need to be scheduled on *m* machines with varying processing power, while trying to minimize the makespan – the total length of the schedule (that is, when all the jobs have finished processing). In the specific variant known as *job-shop scheduling*, each job consists of a set of *operations* O_1, O_2, ..., O_n which need to be processed in a specific order (known as *precedence constraints*). Each operation has a *specific machine* that it needs to be processed on and only one operation in a job can be processed at a given time. A common relaxation is the **flexible** job shop, where each operation can be processed on any machine of a given *set* (the machines in each set are identical).

Job shops are typically small manufacturing systems that handle job production

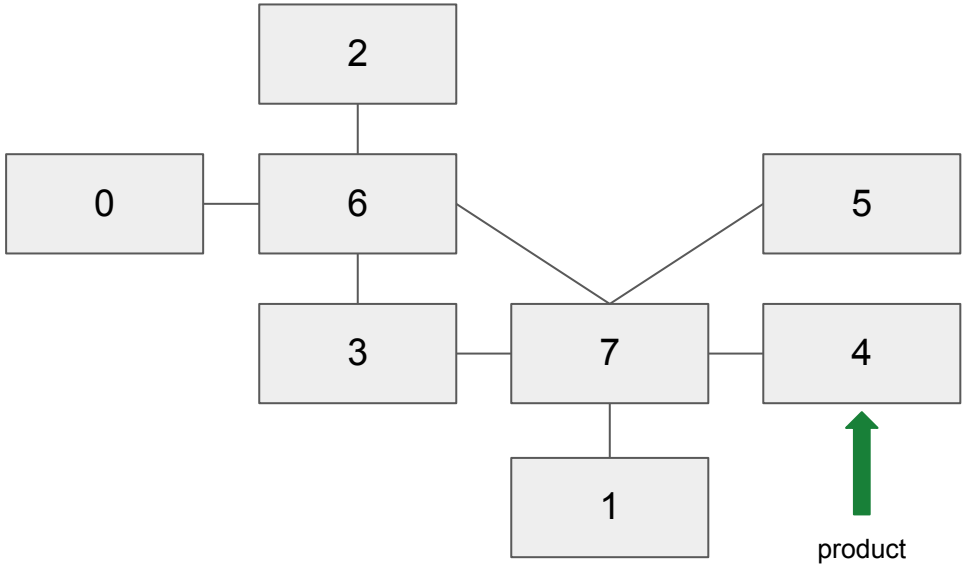
Source: Wiki

Our problem is a **Dynamic Flexible JSP (DF-JSP)** where each job is the production of a product and each operation is a skill to be performed on that product by a specific machine (in a set of feasible machines)
dynamic = on-line scheduling to handle dynamic events such as the insertions, cancellations or modifications of orders, machine breakdowns, variations in processing times and so on...

Job-shop scheduling

Job = { operation_1, operation_2, …, operation_n}

Machines = { machine_1 , machine_2 , …, machine_m}



Possible objectives: minimize duration, energy, etc.

Possible solutions

- **Standard methods**
  Dispatching rules (deterministic rules) and meta-heuristics (genetic algorithms and particle swarm optimization)

- **Innovative possibility**
  Multi-Agent Reinforcement Learning (MARL)

RL problem

Episodic Dec-POMDP

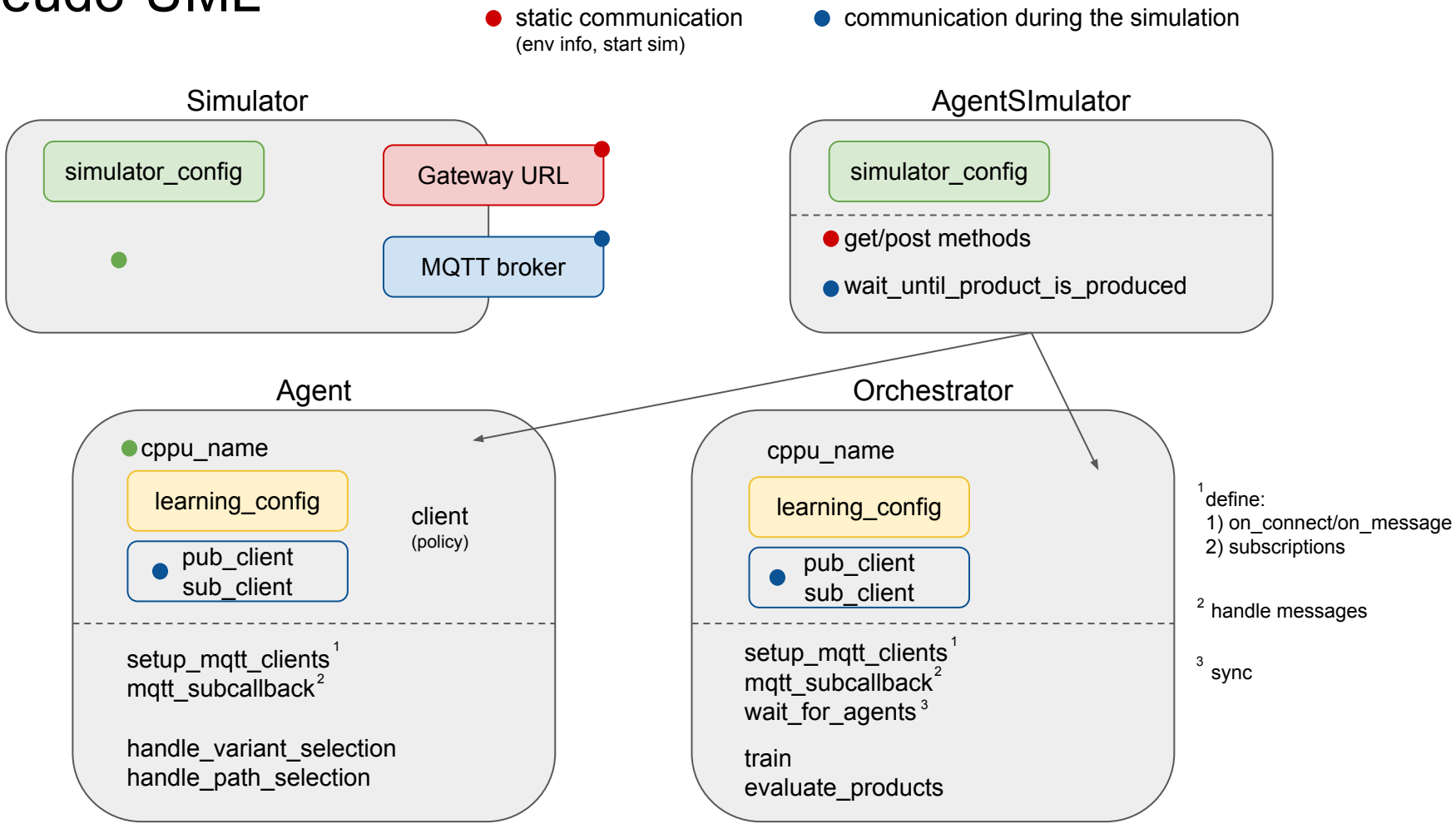
Discounted setting

Decentralized architecture (each agent is independent and has only local information)

Observation = state	Actions	Reward
(required skill, product)	ports	based on the duration of the transport
small observation space: a few skills x a few products	small action space	sparse reward

Partial observability
When there is 1 product only, partial observability becomes full observability if we let the two consecutive agents communicate. (All the other MDPs are static and hence there is nothing to be observed about them)
More than 1 product --> partial observability
Turn-based: 1 action at a time by a single agent (1 product), more than one action at a time by multiple agents (asynchronous setting)

Pseudo-UML


static communication
(env info, start sim)
communication during the simulation
Simulator
simulator_config
Gateway URL
MQTT broker
AgentSImulator
simulator_config
get/post methods
wait_until_product_is_produced
Agent
cppu_name
learning_config
pub_client
sub_client
client
(policy)
setup_mqtt_clients 1
mqtt_subcallback 2
handle_variant_selection
handle_path_selection
Orchestrator
cppu_name
learning_config
pub_client
sub_client
setup_mqtt_clients 1
mqtt_subcallback 2
wait_for_agents 3
train
evaluate_products
1 define:
1) on_connect/on_message
2) subscriptions
2 handle messages
3 sync

MQTT communication

sync

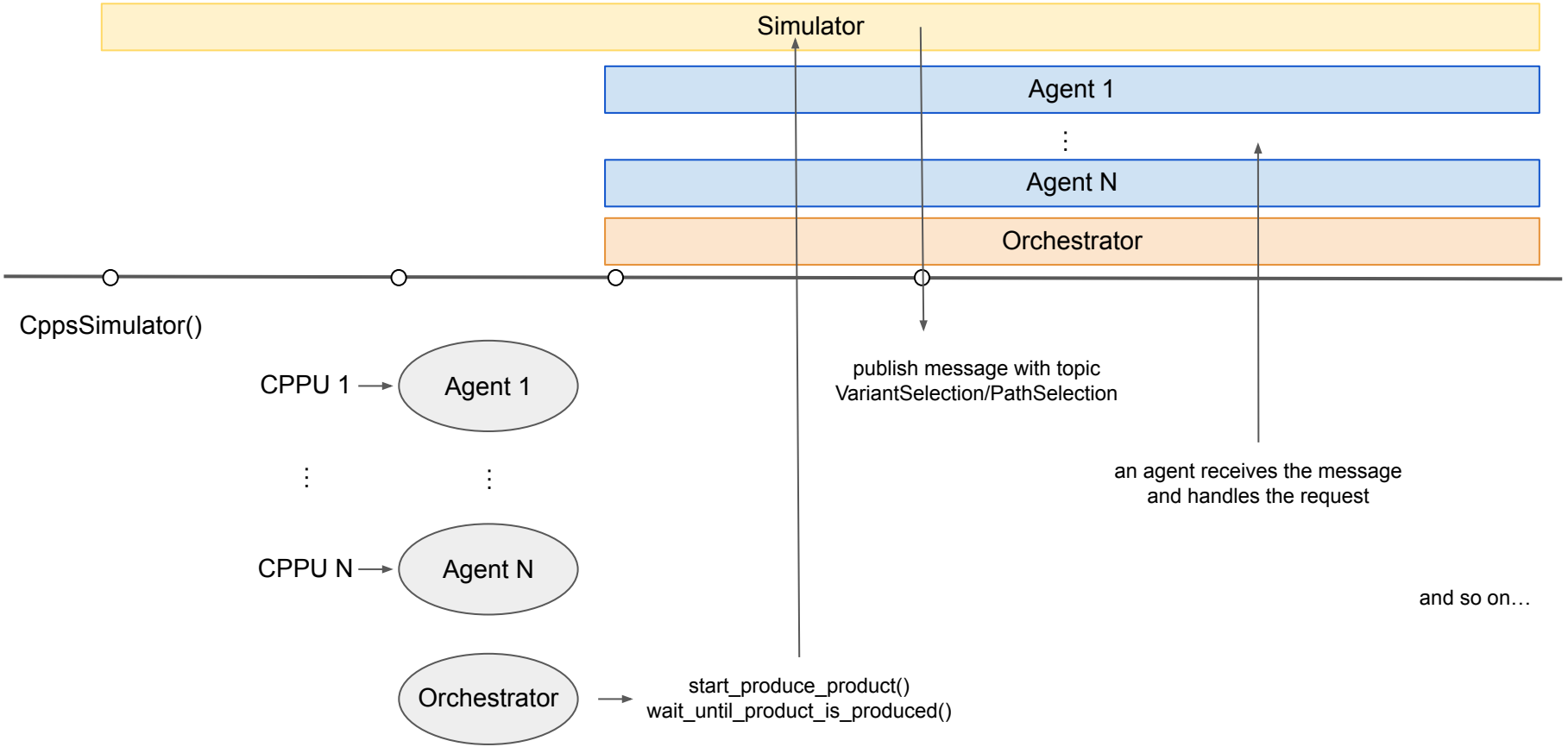
Message	Publisher	Published in	Subscriber	Callback
VariantSelection	Agent	publish_variant in handle_variant_selection	Agent	handle_variant_selection
PathSelection	Agent	publish_port in handle_path_selection	Agent	handle_path_selection
PathReward	Simulator	-	Agent	propagate_reward_to_path_gym
VariantStatus	Agent	publish_variant_learning_status in handle_variant_selection	Simulator	-
PathStatus	Agent	publish_path_learning_status in handle_path_selection	Simulator	-
TrainingRegime	Orchestrator	publish_training_mode in train/evaluate_products	Agent	update_training_regime
EpisodeManagement	Orchestrator	publish_episode_management in train and constructor	Agent	start_episode/end_episode
AgentReadyRequest	Orchestrator	wait_for_agents	Agent	publish_ready
AgentReady	Agent	publish_ready	Orchestrator	remove_cppus

MQTT communication 2

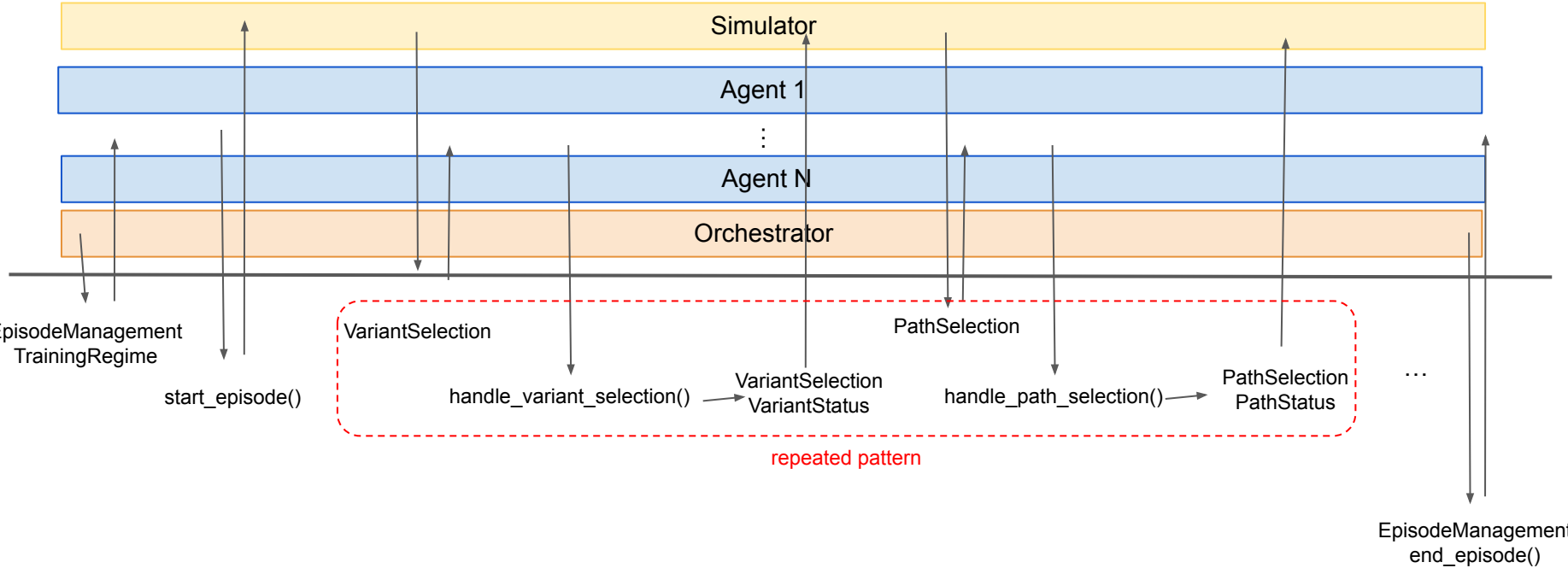
sync

Message	Publisher	Published in	Subscriber	Callback
StartedEpisode	Agent	publish_started_episode in start_episode	Orchestrator	add_cppus
EndedEpisode	Agent	publish_ended_episode in end_episode	Orchestrator	add_cppus
InfiniteLoop	Agent	learned_step in handle_path_selection	Orchestrator (AgentSImulator)	mqtt_client.disconnect()

Distributed architecture (multiprocessing)



Simulation flow



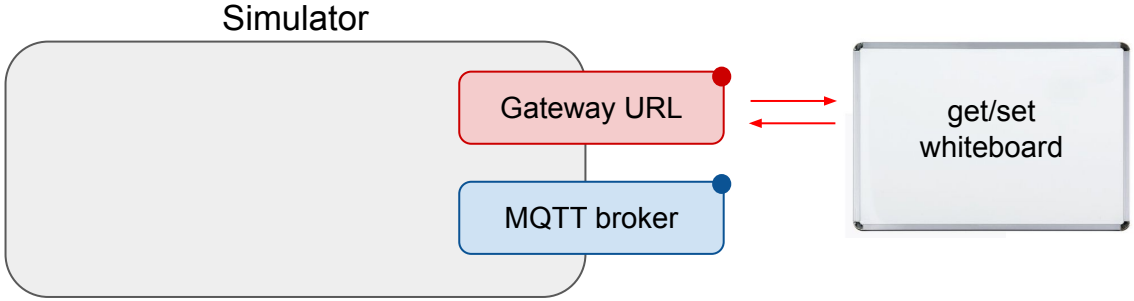
wait_for_agent()
AgentReadyRequest
→
publish_ready()
AgentReady
→
remove_cppus()

Policy for path selection

(Tabular) Distributed Q-learning

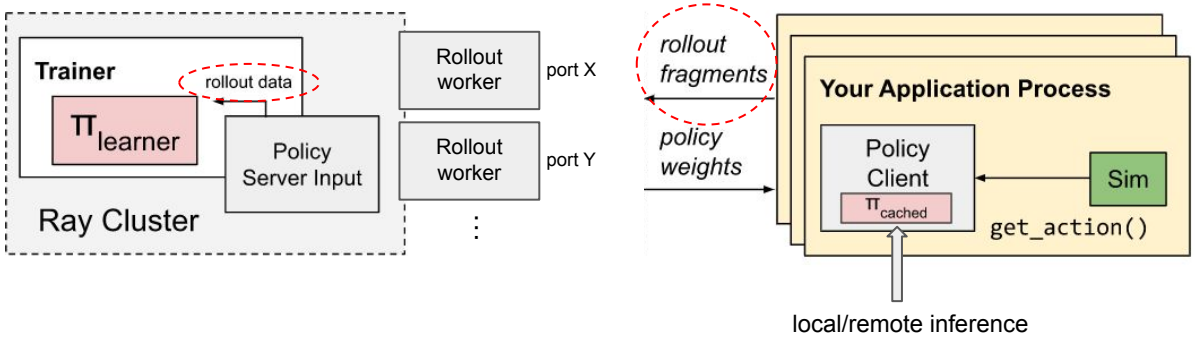
$$Q_i(s,a) = (1-\alpha)Q_i(s,a) + \alpha\left[R(s,a) + \gamma \max_{a'} Q^{\text{next}}(s_{\text{next}}, a')\right]$$

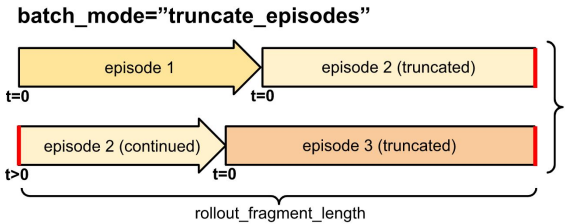
Communication between agents



$s_{\text{next}} = s$ if s is (required skill, product) → the current agent calls handle_path_selection when it cannot perform that skill → the next agent observes the same thing

RLlib client-server





RLlib collects batches of size `rollout_fragment_length` from rollout workers, and concatenates one or more of these batches into a batch of size `train_batch_size` that is the input to SGD.

Rollout = episode --> Each rollout worker creates its own listening socket for incoming experiences (collection of batches) and then forwards the new policy to the connected client

Any number of clients can connect to any number of rollout workers → the learned policy is the same!