Master thesis

Master in Innovation and Research in Informatics (MIRI)

Auto-scaling a video-conference platform with Reinforcement learning

REPORT

Author: Francesc Roy Campderrós

Director: Leandro Navarro (DAC - DSG)

Codirector: Felix Freitag (DAC - DSG)

Date: 18/10/2021

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Abstract

One of the capabilities that video-conferencing platforms are expected to have, as well as other distributed services, is being able to **scale** horizontally. This is because workload is not constant in a lot of applications, so setting a fixed number of servers beforehand will probably end up with either bad quality of service when load is too high, or resources wasted when load is too low. From the service providers's point of view both situations are undesirable. On the one side, they may be penalised when not delivering sufficient quality of service to their users. On the other side, having servers infra-used is inefficient, as more servers running imply higher electricity/renting costs. Therefore this auto-scaling capability is crucial in order to optimize the expenses at the end of the month.

In this work we develop an auto-scaling algorithm based on Reinforcement learning (RL) to be applied to the adjustment of computing capacity of a distributed video-conference platform such as Jitsi and perform a comparison with simple threshold based methods (TBM), which are offered by many cloud providers as the default auto-scaling service. We perform this comparison under different synthetic workload patterns. Since video-conferencing platforms consume a lot of computing resources and we want to analyse different high loads, the comparison is done with simulations.

We demonstrate that RL performs better than TBM in all the scenarios evaluated in terms of money expended (with different patterns tested) and that the difference between them is accentuated the more complex the workload pattern is.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Videoconferencing has experimented a huge growth during last years becoming an important medium in our everyday lives. From working to education to even socializing, we are moving to video-conference platforms. This has been possible due to the users having more powerful personal computers and high network bandwidth offered by network providers, given that videoconferencing is traffic intensive [1].

The pandemic has only exaggerated this trend and it seems that public governments and private companies are planning to keep with this tendency after it finishes, suggesting their employees to do remote work 2 or 3 days per week as it seems an effective formula.

One of the capabilities that videoconferencing platforms are expected to have, as well as other distributed services, is being able to scale horizontally, that is, increase (scale out) the number of servers attending requests if we have a bigger input load in a certain moment or decrease (scale in) them if the load goes down again.

This is because as a service provider we want to offer a good service quality (at least as good as the service level agreement or SLA [1]) to our users, but always having in mind that more servers up implies higher costs (electricity or renting costs depending if we own the machines or not) at the end of the month. This automatized scaling actions are a procedure that administrators of these platforms try to optimize.

Thus, auto-scaling is a highly interesting topic for the distributed systems research community. There exist different auto-scaling techniques varying from more reactive to more proactive ones.

[1]SLA is the limit acceptable by the user for a monitored metric of the system, for example: 70% CPU load, 35 packets/second lost, 20 ms response time,etc. An economic compensation may be required when violated.

1.2 Problem statement

Having in mind that a server running has a price per second and not complying with an SLA agreement with users also has a (different and usually higher) price per second, the problem to solve is to find the optimum number of servers in each moment in terms of money expended (€).

Therefore, between 2 auto-scaling algorithms that switch on and switch off servers of a distributed platform, we will consider better the one that implies a lower cost at the end of a certain deployment period.

The basic idea of any auto-scaling algorithm is the following: On the one hand if the current running servers are highly loaded and we cannot offer good enough service quality to our users, a new server will be booted up and new video-conferences will be placed to that new server [2]. On the other hand if some servers are infra-used, and we are able to deliver a sufficient service quality with less resources, some of them will be switched off.

Depending on the technique used to develop the algorithm the quality of the solution in terms of expenses will be better or worse.

It is worth mentioning that the problem of auto-scaling in videoconferencing services is a little bit different than the same problem in other distributed systems like distributed database management system or a "distributed web server" (a group of web servers hosting the same website). In video-conference platforms, jobs (video-conferences) performed by the servers last from minutes to hours. In contrast, jobs (queries or web page delivery) in the mentioned above distributed systems last some orders of time magnitude less. Moreover, in a videoconferencing platform if some server is switched off, the running conferences inside it have to be reallocated (in a distributed web server or database management system, the system waits until the running requests are finished before switching off).

1.3 Contribution

The contribution is the development of an auto-scaling algorithm based on Reinforcement learning theory. Once implemented we compare it to a simpler Threshold-based rules technique as the base case. The comparison is done using a simulator and under different fictitious workload patterns, and we observe which algorithm reacts better in terms of overall costs.

In the future (not implemented yet) we are going to repeat the comparison in a real environment built in the guifi.net testbed [2]. The video-conference platform to do the real test will be Jitsi [3], an open source platform. Workloads will be fictitiously generated with *Selenium*, a browser automation framework (but based on real observed traffic from a Jitsi System that is already in production in the mentioned testbed).

The idea is that with this implementation we are going to propose improvements over the current solution for auto-scaling that Jitsi is offering its users [4].

It is important to notice that in this work we are only going to consider scaling in and out.

[2] Usually there is a load-balancer in front of the servers that distributes the incoming load basing its decisions on the current load of each server.

Another approach could had been scaling up and down, that is resizing the resources assigned to a single virtual machine (VM). For example if a VM has been provisioned with two vCPUs and more load is coming, the hypervisor could momentarily increase to three vCPUs.

1.4 Context

This work was carried out under the supervision of Leandro Navarro and Felix Freitag on the context of the LeadingEdge project, funded by the Chistera initiative [5].

1.5 Thesis Outline

This thesis is organized as follows. The background and related work is reviewed in the next chapter, Chapter 2. Chapter 3 describes the system and the problem in more detail. Chapter 4 presents the modeling of our problem in the context of Reinforcement Learning. In Chapter 5, we present the results. Finally, the last chapter concludes this thesis.

2 Background and Related work

2.1 Related work

There exist a lot of papers related to the auto-scaling problem. Some of them try to classify the different approaches that exist to solve it. They are mainly 5 categories: static threshold-based rules, control theory, reinforcement learning, queuing theory and time series analysis.[6]

Most of them are based on the MAPE (monitor, analyze, plan and execute) loop [6]:

- Monitor is about taking an snapshot of the desired metrics (CPU usage, drop packets per second, latency of requests, etc.) of the system at a suitable granularity (every one second, 5 seconds, 1 minute...). This metric is the mean considering all the servers. Usually the provider of the VM's will offer an API to connect to the hypervisor and gather this information.

- Analyze consists of processing the previous metrics gathered and then getting useful information of the system utilization (and maybe also system utilization in the future if they do prediction).

- Plan is the phase where the auto-scaler designs the actions to take based on the previous results of the analyze phase.

- Execute is where the actions designed in the previous state are actually executed through the API that the provider expose to us to be able to talk to the hypervisor.

It is important to distinguish between auto-scaling state-based distributed systems or stateless ones, because the solution should take into account additional considerations if we are dealing with an state-based (because the state should be replicated (or transferred) in the new server). An example of state-based distributed system could be a distributed database management system and an example of a stateless could be a "distributed web server". In this work we deal with a stateless distributed system.

From the 5 techniques mentioned above some of them are more reactive (they react after they capture some event) and the other more proactive (they can anticipate and react before some event has happened).

If we focus in the most simpler reactive solutions, we find a lot of commercial tools that use this approach. For example Amazon Web Services (AWS) use threshold based methods as its default auto-scaling service [7]. Also it is the way that RightScale, a popular service in cloud infrastructures, operates [8].

If we focus on solutions that are a mix of reactive and proactive approaches we find the work of Ying Liu et al. named ProRenata [9]. ProRenata is an auto-scaler for a Cassandra-like distributed database management system that explicitly considers scaling overhead, i.e., data migration cost, to achieve high resource utilization and low latency SLA violation. ProRenata achieve better results than its proactive competitors because it adds a reactive module that allows it to compare the prediction at "t-1" to the output of the reactive module at "t" and perform adjustments if needed (usually there is error in the prediction).

Also in the industry this kind of approaches mix is found. For example Netflix, whose servers

are hosted in AWS, use AWS reactive auto-scaling system in combination with a internal predictive model (based on Fast Fourier Transform) [10][11].

Nevertheless in this project we focus our attention in one particular approach, Reinforcement learning, that is a pure proactive approach. A lot of work has also been done in this field [12]. For example Dutreilh et al. in [13] model the problem as an MDP with a state space consisting in 3 dimensions (workload in number of requests per second, the current number of running virtual machines and the performance expressed as the average response time to requests in seconds). They use Q-learning [14] to find a good policy. Their reward function inspired the reward function of our work although they penalize the action of booting a VM. They use fictitiously generated workload to test their solution in a real testbed. They use advanced techniques (initialization of the Q function, convergence speedups, performance model change detection system...) to overcome the problems of having good policies in the early phases of learning, time for the learning to converge to an optimal policy and coping with changes in the application performance over time.

There are other works that mix different techniques mentioned above. For example Dezhabad et al. in [15] combine RL with a genetic algorithm and queue theory to auto-scale virtualized firewalls. Barret et al. use parallel executions of diferent autoscalers in order to speed up the training phase [16].

There exists also a lot of work regarding using fuzzy logic (that falls inside control theory), that consists on using more sophisticated rules instead of the typical threshold based (qualitative instead of quantitative rules). For example Jamshidi et al. in [17]. developed RobusT2Scale, an autoscaler with this technique (and time series analysis) and proved that it is significantly better than simpler threshold based ones.

On top of that there also exist a lot of work regarding mixing fuzzy logic and reinforcement learning. The idea is to find the optimal rules through interacting with the system. Here we can find a lot of work well summarized in [18].

2.2 Jitsi architecture background

In the following figure 1 we can see the basic architecture of the video-conference system that we are going to deal with:

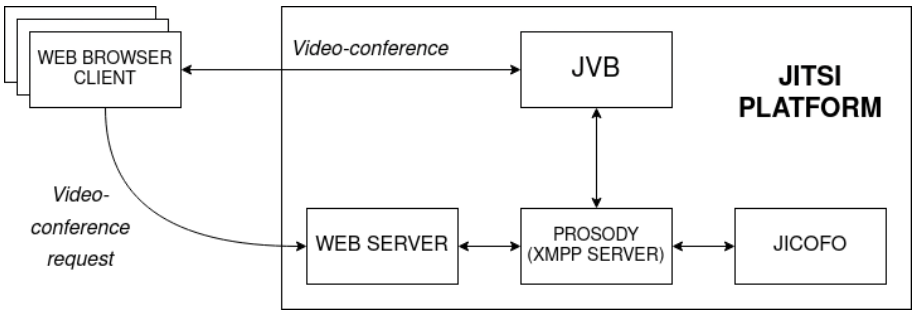

Figure 1: Jitsi architecture

This is the basic architecture with just one server holding conferences.

Let's describe the different components of Jitsi:

- The Web Server (usually *Nginx*) serving the front-end files of Jitsi.
- Jitsi Video Bridge (*JVB*) is the actual video-conference server, that is, a Selective Forwarding Unit that receives audio and video streams from endpoints and relays them to everyone else. It does what a Multipoint Control Unit used to do in old video-conference systems but with an improved philosophy (no mixing of streams) [19]. This is actually the component that can be **scaled**.
- Jicofo is the conference focus agent, that is, it is responsible for managing media sessions between each of the participants and the JVB, i.e, signaling. It is also responsible to select the JVB that is going to host the new conference (round-robin as default but also configurable to less loaded JVB).
- Prosody is an XMPP Server that allows the communications between all components with XMPP messages.

This would be the default installation of Jitsi in a single server. However Jitsi is ready to grow horizontally, i.e to **scale**, by adding more JVBs in the same machine or another one (as many as the administrator needs). The only necessary thing would be to indicate them where Prosody resides and announce themselves. Jicofo would then notice new JVBs instances and would consider them when dealing with new conferences. If a server is switched off and there are live conferences running inside it, Jicofo will reallocate these conferences (transparently for the user) to available servers. This combination of Jicofo, Prosody, Web server and JVBs is what is described as a Jitsi shard.

Clients are browsers with WebRTC [20] technology. Most major browser such as Chrome, Firefox, Safari and Opera have this feature implemented.

In the default installation of Jitsi, a specific video-conference (so all its participants) can be hosted only by one JVB. This means that it is not possible to allocate some participants in one JVB and some other participants of the same conference on another. However this can be configured, but first let's describe how a more complex Jitsi deployment would look like.

Suppose we are offering videoconference service around the world. We would like to have different Jitsi shards distributed around different geographical areas and this way users of the same conference (typically of the same geographical area) would be allocated to the same shard (with the help of a unique load-balancer that would redirect the user to its nearest shard). We will benefit of this fact because users would be closer to the server (less latency).

This a perfect solution if you think there is not going to be a lot of international conferences in your Jitsi system. However there is a drawback with this implementation. Suppose there is video-conference with users from different geographical areas. The user that creates the room is from Australia, so the load-balancer redirects her to its nearest shard. But after that when new users (from USA for example) want to join the room, the load-balancer will redirect them to the same shard as the Australian user:

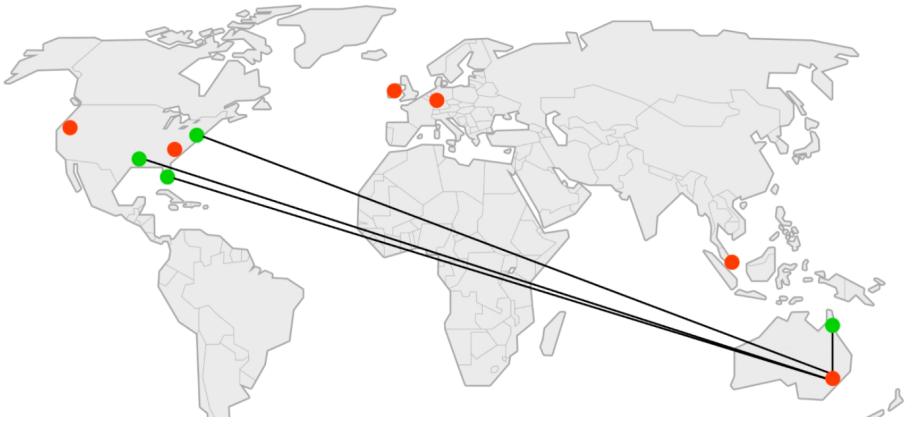

Figure 2: Jitsi default with serveral shards

So this situation in figure 2 is not a good design because it is possible that some USA users have high latency to the JVB and this will downgrade the video-conference. So the solution is allowing video-bridges to talk between them (using a protocol named OCTO):

Figure 3: Jitsi with OCTO enabled

Where the red points represents the shards (Jicofo and pool of JVBs) and green points the users.

So the idea now is that even the signaling procedure will be handled by the shard located in Australia again, its Jicofo sees now all JVBs from all shards as possible JVBs to allocate users, so North-american users will be allocated to JVBs from the shard located in USA and the Australian user in the Australian shard. JVBs will then be talking to each other to allow the communication (possibly through a low latency channel provisioned by a company).

OCTO can also be enabled even if only one shard is available. It is a way to allow Jicofo to allocate users of the same conference to different JVBs. This is how we are going to configure our infrastructure.

2.3 Threshold based methods (TBM) background

There is not a lot of theory needed in order to implement a TBM method. An example in pseudo-code of a threshold method could be:

```
Loop forever:
    Monitored metric = Take snapshot of the system()
    if(Monitored metric > Upper threshold):
        Add a server()
        Wait COOLDOWN seconds()
    else if(Monitored metric < Lower threshold):
        Stop server()
        Wait COOLDOWN seconds()
    else:
        pass
  Wait PHOTO INTERVAL second() # monitoring polling strategy...
```

Table 1: Threshold based algorithm

The main idea here is to maintain the monitored metric between two values (Upper threshold and Lower threshold). The monitored metric could be for example: **mean** current load of the servers, **mean** current packet loss rate, **mean** current response time, etc.

Upper threshold will be below and presumably close to SLA agreement and lower threshold will be bellow Upper threshold. COOLDOWN can be 20 seconds, 30 seconds, 2 minutes... It should be a time that is enough to see the effects of actions we have performed to the system (this is usually called "cooldown period") to avoid oscillation of resources.

The advantage of threshold based methods is their simplicity in terms of implementation. The downside of threshold base methods is the difficulty in setting good thresholds.

As an example, suppose the load is growing and Upper threshold is reached. If you set the thresholds too close to the SLA agreement, there exists a higher risk to surpass the SLA agreement because we will react too late if the load keeps growing (because as we already said booting up a server takes time). So probably we won't be delivering a sufficient quality during some time (and thus paying) until the server is eventually up.

Of course a possibility could be to increase the distance from thresholds to SLA agreements in order to avoid this situation, but maybe you would be anticipating too much and thus expending money unnecessary.

Even if one manages to find the optimal thresholds, they have to be set for a particular workload. If the load of the service changes over time, the algorithm won't be able to adapt itself to the new situation.

2.4 Reinforcement learning theory (RL) background

In this work we develop a more intelligent auto-scaler based on reinforcement learning that ideally will perform better and give better results in terms of cost (€), having in mind that a virtual

machine running has a price/second and violating SLA agreements also has a price/second.

Reinforcement learning theory is build on top of a Markov[3] Decision Process (MDP) concept. A MDP is a mathematical concept to model a real process where there is an (intelligent) agent that can map the world state on an internal simplified state and then interact with it. It will interact with it taking actions (through rounds) that actually are going to change that state.

Each action that the agent performs (from a set of possible actions), as already said, modifies the state of the world, so it can help the agent to get closer to a desired state or vice versa.

Due the stochasticity of the world, given that the agent is in a state and takes an action, it can end in different states with different probabilities.

To model the quality of an state a reward (or utility) function is defined. The reward function is something that the developer should design carefully, taking into account that this measure will obviously influence when searching for best actions to perform from each state.

Recapitulating, our agent is going to take different actions (one on each round) and transitioning from state to state through time. The mapping of each state S to an action A is called a policy Π.

$$\Pi : S \longrightarrow A(s) \tag{1}$$

One could then suspect that there are exactly $|A|^{|S|}$ possible policies.

Starting from a start state, if we follow a policy as an agent, we will navigate through the state space (this is an **episode**) until an end state (or forever if there is no end state). One could compute how good a policy is summing all the rewards received in an episode. However even with a specific policy, the agent can take different episodes (stochasticity of the world) so a better way to evaluate the goodness of a policy is computing the expected reward.

So the objective is to find the optimal policy or at least a good policy, that is, a policy that has a high expected reward.

[3] As the name states, a Markov decision process has to hold the Markov property, i.e, given information of the current state and an action is sufficient to know the future (it is independent of past states or actions that lead you up to here).

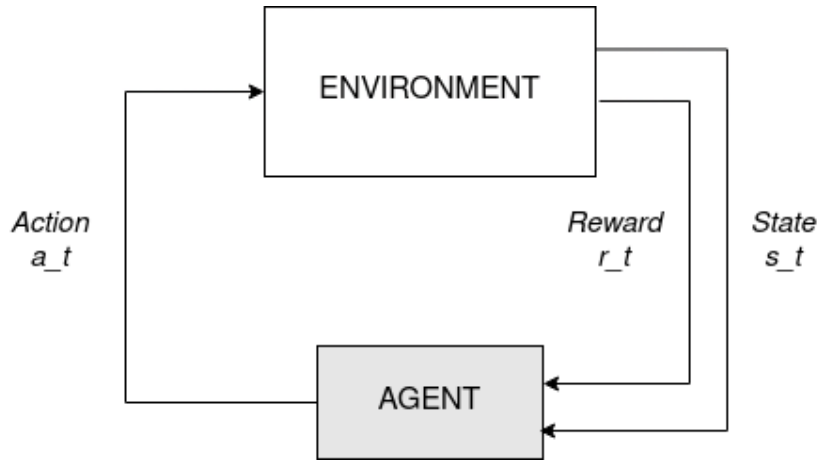

Figure 4: MDP scheme

2.5 Reinforcement learning toy problem

As an example, consider a simple MDP which simulates a robot (the agent) interacting with the world with five possible actions: one step north, one step south, one step east, one step west or do nothing. Here the world state is modeled with just the x position and the y position of the robot (7 x 7, 2 dimension grid), so we have 49 states.

This toy problem will help us to later formulate the real auto-scaling problem.

Definitions:

- $States$: the set of states (each position of the 2D grid in the example)
- $S_{start} \in States$: starting state
- $Actions(s)$: possible actions from state s (N,S,W,E,*Nothing* in the example)
- $IsEnd(s)$: whether at end state or not
- $T(s, a, s')$: transition probability from s to s' if taking action a. This is also called the dynamics. We are going to suppose that the dynamics of the world do not change over time (they are stationary).
- $Reward(s)$: reward for being at state s. There are other ways to define the reward, for example as $Reward(s, a)$, that is, the reward is for being at state s and taking action a.

When the robot performs its action it has 75% of probability of performing it correctly, 15% probability of probabilities of malfunctioning and perform it in a 90 degrees direction and 15% of probabilities of malfunctioning and perform it in the other 90 degrees direction (these are the transitioning probabilities) for N,S,W or E. For the action *Nothing*, the probabilities of performing the action correctly are 80%. Then it has 5% probabilities to go to any direction (for example because of the wind hitting our agent).

There will be one end states in our MDP, $(x, y) = (3, 3)$ (there could be more or any). And as an example we would like that our agent tries to reach state $(3, 3)$ as fast as possible. To achieve

that, first we define the reward function for the state as follows:

$$Reward(s, a) = -[(x-3)^2 + (y-3)^2] - STEP\_COST$$

With STEP_COST = 0 if a is "Nothing", 0.01 otherwise

So implicitly we are telling our agent that actions that take him closer to (3,3) are more beneficial. Pictorially we could draw our MDP like the following image 5.

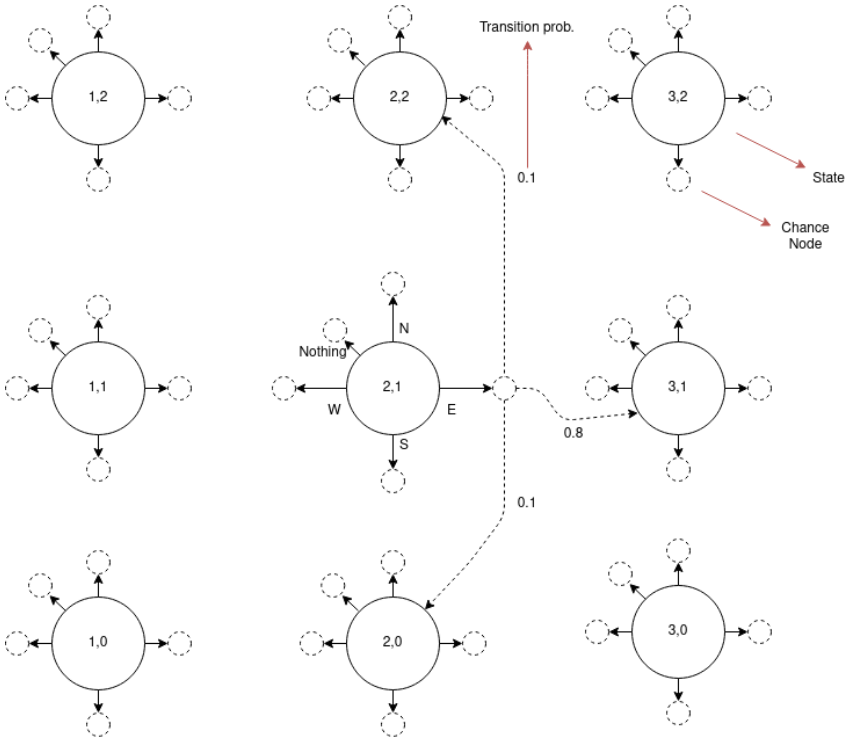

Figure 5: MDP toy problem

In figure 5 we have only drawn part of MDP, there are states missing and only the transition probabilities of one chance node of state 2,1 has been written as an example.

An example policy could be (not specifically the best one):

N	S	E	N	N	N	N
S	N	S	N	N	N	N
E	E	E	W	N	N	N
N	N	N	·	N	N	N
N	N	N	N	N	Nothing	N
N	N	N	N	N	N	N
N	N	N	N	N	N	N

Table 2: Policy example

Where "·" means no action because it is an end state.

An example of an episode (sequence of actions until end state) following this policy could be (starting from 2,2):

$$(2,2) \longrightarrow (2,3) \longrightarrow (2,4) \longrightarrow (3,4) \longrightarrow (3,3) \tag{2}$$

Note that the agent malfunctioned (luckily) in the fourth step.

We could compute the reward of an episode like this:

$$Reward = Reward(s_t) + \gamma * Reward(s_{t+1}) + \gamma^2 * Reward(s_{t+2}) + ... \tag{3}$$

Where γ is the discount factor that we can fix between $[0 - 1]$. The meaning of setting γ to 0 is that we only care for immediate rewards. On the other hand, a γ value of 1 that we care the same for immediate reward than for future very distant rewards. Usually γ is set around 0.9.

But it would be more useful to compute the expected reward (also called **Value**) of a policy Π from a particular start state s:

$$V_\Pi(s) = \mathbb{E}_\Pi[Reward] \tag{4}$$

Lastly, one interesting concept that one could want to compute and is going to be useful later is $Q_\Pi(s,a)$, that is the expected reward following the policy Π but after taking action a instead of $\Pi(s)$ in the first move.

2.6 Reinforcement learning algorithms for model-based MDPs (dynamics known)

In this situation, when the transition probabilities are known, there exists algorithms to compute exactly the value of a policy and to find the optimum policy.

2.6.1 Algorithms for policy evaluation

There exist an iterative algorithm (based on Dynamic programming) called Policy evaluation that computes the expected reward of a given policy Π from each state:

Initialize $V_\Pi^{(0)}(s) \longleftarrow 0$ for all states s
While $(||V_\Pi^{(t)} - V_\Pi^{(t-1)}|| \geq \epsilon)$:
    For each state s:
        $V_\Pi^{(t)}(s) \longleftarrow Reward(s, \Pi(s)) + \gamma * \sum_{s'} T(s, \Pi(s), s') * V_\Pi^{(t-1)}(s')$

Table 3: Policy Evaluation algorithm

It is also possible to compute the expected reward analytically solving a system of Bellman's equations:

$$V_\Pi(s) = R_\Pi(s) + \gamma * \sum_{s' \in S} P_\Pi(s'|s) * V_{\Pi(s')} \qquad \forall s \in S \tag{5}$$

2.6.2 Algorithms for control

One could also want to compute the optimum policy Π_{opt}, i.e, the one that give us the highest expected reward with this algorithm:

Set $i = 0$
Initialize $\Pi_0(s)$ randomly for all states
while $i == 0$ or the policy has changed in any state:
    $V_{\Pi_i} \longleftarrow$ policy_evaluation(Π_i)
    $\Pi i + 1 \longleftarrow$ policy_improvement(V_{Π_i}, Π_i)
    $i = i + 1$

Table 4: Policy Iteration algorithm

For s in S an a in A:
    $Q_{\Pi_i}(s, a) = R(s, a) + \gamma * \sum_{s' \in S} P(s'|s, a) * V_{\Pi_i}(s')$
For s in S:
    $\Pi_{i+1}(s) = argmax_a Q_{\Pi_i}(s, a)$

Table 5: Policy Improvement function

Remember that the expression that we find in this function $Q_{\Pi}(s, a)$ has an important meaning as stated before in subsection 2.5.

It's easy to see that the best policy of this toy problem will be (and in fact is what 9 give us):

S	S	S	S	S	S	S
E	S	S	S	S	S	W
E	E	S	S	S	W	W
E	E	E	·	W	W	W
E	E	N	N	N	W	W
E	N	N	N	N	N	W
N	N	N	N	N	N	N

Table 6: Best policy

2.7 Reinforcement learning algorithms for model-free MDPs (dynamics unknown)

In the unrealistic setting explained above (where we know how the world works and are able to write down the probabilities of an action taking the agent to specific state) we could compute the expected reward given a policy using the algorithm stated above 3. Furthermore we could find the best policy, i.e, the one giving us the highest expected reward 9. In this scenario we say that we have a model-based MDP.

Nevertheless when we develop an agent (the auto-scaler in our case) that is going to interact with the world (the infrastructure consisting of the servers,etc) we don't know a priori the transitioning probabilities so we are not able to compute the expected reward given a policy neither

the best policy one can obtain. So we have a model-free MDP.

So here is where RL comes in. The idea behind RL is that the agent is going to figure out the expected reward of a policy (and also find the optimal policy) from experience. That is, interacting with the world we are going be able to compute estimations of $V_{\Pi}(s)$ and $Q_{\Pi}(s,a)$ by indirectly computing estimations of the transition probabilities $T(s,a,s^{'})$ of the world.

2.7.1 Algorithms for policy evaluation

The first Naive algorithm that we can think of is a Monte-Carlo algorithm. This algorithm gives us an unbiased estimator of $V_{\Pi}(s)$ and it works by performing a huge number of episodes and then computing the average of the reward obtained on each of them:

Initialize $N(s) \longleftarrow 0$, $G(s) \longleftarrow 0$ for all states s
$N(s)$ is a counter of total first visits, $G(s)$ is total reward
Loop:
    Sample episode $i = s_{i,1}, a_{i,1}, r_{i,1}, s_{i,2}, a_{i,2}, r_{i,2}, ..., s_{i,T_i}$
    Define $G_{i,t} = r_{i,t} + \gamma * r_{i,t+1} + \gamma^2 * r_{i,t+2} + ... + \gamma^{T_i - 1} * r_{i,T_i}$
    # $G_{i,t}$ is the reward from time step t onwards in ith episode
    For each state s visited in episode i:
        For first time t that state s is visited in episode i:
            Increment counter: $N(s) = N(s) + 1$
            Increment total reward: $G(s) = G(s) + G_{i,t}$
            Update estimate: $V_{\Pi}(s) = G(s)/N(s)$

Table 7: Monte Carlo algorithm

A drawback of this algorithm is that only works with MDPs that have at least one reachable end state in contrast of Policy Evaluation 3 or algorithms that we are going to see below.

So we need an algorithm that can find the expected reward of a given policy from a model-free MDP even if there is no end state. This is temporal difference learning (TD-learning):

Initialize $V_{\Pi}(s) = 0, \forall s \in S$
Loop:
    Sample tuple$(s_t, \Pi(s_t), r_t, s_{t+1})$
    $V_{\Pi}(s_t) = V_{\Pi}(s_t) + \alpha * ([r_t + \gamma * V_{\Pi}(s_{t+1})] - V_{\Pi}(s_t))$
    $s_t = s_{t+1}$

Table 8: TD learning algorithm

Where α is a learning rate that can be constant (should be small if we don't want the values to oscillate and never converge) or we can make it lower as we make more iterations of the loop.

Note that with this algorithm there is no need to wait until the end of an episode to update our $V_{\Pi}(s)$.

The estimator that this algorithm is going to give us is biased but it has less variance that the one obtained with 7.

2.7.2 Algorithms for control

A first algorithm for control (finding Π_{opt}) in a model free MDP is the following Monte Carlo method which has a similar structure than Policy Iteration 9 but here we will obtain an estimate of $Q_\Pi(s,a)$ instead of the real $Q_\Pi(s,a)$:

Initialize $N(s,a) = 0$,$G(s,a) = 0$,$Q_\Pi(s,a) = 0$, $\forall s \in S\ \forall a \in A$
Set $i = 0$
Initialize $\Pi_0(s)$ randomly for all states
while $i == 0$ or the policy has changed in any state:
    $\hat{Q}_{\Pi_i}(s,a) \longleftarrow$ compute_estimate_of_Q_PI(Π_i)
    $\Pi i + 1 \longleftarrow argmax_a \hat{Q}_{\Pi_i}(s,a)\ with\ prob\ (1-\epsilon)\ or\ random(Actions)\ with\ prob\ \epsilon$
    $i = i + 1$

Table 9: Monte Carlo Control algorithm

Loop:
    Sample episode $i = s_{i,1}, a_{i,1}, r_{i,1}, s_{i,2}, a_{i,2}, r_{i,2}, ..., s_{i,T_i}$
    Define $G_{i,t} = r_{i,t} + \gamma * r_{i,t+1} + \gamma^2 * r_{i,t+2} + ... + \gamma^{T_i - 1} * r_{i,T_i}$
    # $G_{i,t}$ is the reward from time step t onwards in ith episode
    For each state (s,a) visited in episode i:
        For first time t that (s,a) is visited in episode i:
            Increment counter: $N(s,a) = N(s,a) + 1$
            Increment total reward: $G(s,a) = G(s,a) + G_{i,t}$
            Update estimate: $Q_\Pi(s,a) = G(s,a)/N(s,a)$

Table 10: Compute estimate of Q_Π function

Where ϵ is a decaying parameter that will allow us to explore different state-actions. This is to try to avoid ending with a policy that is actually a local optimum.

However this algorithm has similar drawbacks than Monte Carlo method for policy evaluation, i.e. , we need an end state or this will not work. So a new algorithm is needed and this is where Q-learning comes into action:

Initialize $Q(s,a), \forall s \in S, a \in A, t = 0$, initial state $s_t = s0$
Initialize random policy Π
Loop:
Observe tuple (s_t, a_t, r_t, s_{t+1})
$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha * (r_t + \gamma * max_a Q(s_{t+1}, a) - Q(s_t, a_t))$
$\Pi(s_t) = argmax_a Q(s_t, a)\ with\ prob\ 1 - \epsilon,\ else\ random$
t=t+1

Table 11: Q-learning algorithm

It is worth saying that it is not mandatory to initialize the $Q(s,a)$ function to 0. If we try another initialization we will probably end up with another policy. Setting Q to high values maybe performs better (it is an heuristic).

2.8 Remarks

When dealing with a model-based MDP, we will be able to get the exact solution in both problems. However with model-free MDPs we will only be able to find approximations to the exact solutions of both problems.

3 Description of the system

We design a simulator as real as possible in order to be able to execute the two algorithms (TBM and RL) against it and compare them. But before we compare them we must train the RL algorithm (also against the simulator) in order obtain a good policy.

Once the RL algorithm has been trained (and we have good policy, hopefully a good one, probably not the optimal one) we will then execute both algorithms during a certain amount of simulated time (a week) and see which performs better in terms of money expended 3.2.

In future work, (after the training and comparison against the simulator has been done) we will also do the comparison in the real scenario and see if the same results hold (importing the policy from the previous step for the RL algorithm). If the simulator is a tight approximation of the real system, the results will hold.

The usage of a simulator has been decided because trying to train the RL algorithm in the real scenario is infeasible as it would take too much physical time until a good policy is obtained (a huge number of interactions are needed to get a good estimation of the $Q(s, a)$ function in the Q learning algorithm). This is a drawback of our algorithm in comparison with the TBM as this one is ready to be deployed without any training.

A possibility could had also been to do a first training in the simulator and once the policy is deployed, keep training it in an online manner to get a more accurate policy (according to the real system instead of the simulator), but this has not been implemented due to time restrictions.

In a **first** version we are going to do the comparison under different fictitious traffic connection patterns that repeat themselves in cycles of 10 minutes (with some randomness, that is, the video-conferences will start at same second of the cycle but a normal random variable will be added on each cycle for every participant). The duration of the connections will also be randomized with a new normal random variable on each cycle (the mean will be video-conferences of 2 minutes). This is to emulate the stochasticity of the environment.

In a **second** version we are going to do the comparison also under different fictitious traffic connection patterns but in this case the cycle will be one hour (also with randomness) because is a more realistic scenario. Probably the more realistic scenario would be cycles of one day or one week as this is common in schools (e.g, Maths lesson every Monday at 8:00) or offices (e.g, meeting on Thursday at 11:00 every week).

The TBMs are implemented with different versions (different thresholds) and we will observe if RL is able to improve the performance over all of them.

3.1 Design of the simulator

The design of the simulator [21] tries to be as faithful as possible to the real system thanks to the monitoring implemented using Graphana software [22].With this software we have been able to analyze how the system reacts to every new connection/disconnection in terms of CPU usage, etc and map it to the simulator.

The following classes have been implemented in order to built the simulator:

User class

This class will have 3 attributes:

- id: to uniquely identify it
- type: 1 or 2. Type 1 represents the user without camera/audio. User type 2 a user with camera and audio connected.
- duration: the number of seconds left until disconnection of the conference

This class does not have methods.

JVB class

An object from this class, represents a server.

This class will have 3 attributes:

- up: True if up, False if not
- cpu_load: None if not up, current cpu_load if up
- users_connected: list of User's that are currently being served by this jvb.

This class will have 5 methods:

- start(): sets cpu_load to 0 and up to True
- close(): sets cpu_load to None, up to False, and clears users_connected
- is_up(): returns True if jvb is up, False otherwise
- add_user(user): adds the user to users_connected and increase the cpu_load by +1% (if user of type 1) or +5% (if user of type 2). This is a measure that has been empirically observed in the real system.
- advance_round(): advances one second the simulation for that jvb by decreasing 1 unit on each duration attribute of User's connected. If the User's duration becomes 0, it is removed from the list and the attribute cpu_load is decreased as well.

Jitsi class

An object from this class, represents the entire system.

This class will have 2 attributes:

- video_bridges: a list of available JVBs (up or down).
- video_bridges_up: number of JVBs up

The most important methods of this class will be:

- start_jvb(): it starts a new jvb and increments video_bridges_up attribute after 15 seconds (this is to emulate the behaviour VM machines have when booting up)
- add_user(User): it adds the User to the least loaded jvb from video_bridges list
- stop_jvb(): it stops (also after 15 seconds) the least loaded jvb and reallocates all its users (in case there is any) to other JVBs that are running. It also decreases video_bridges_up attribute
- advance_round(): it advances round for all the system, that is all running JVBs
- get_state(): it returns the current state of the system as a tuple composed by the number of running JVBs, second of the cycle and mean CPU load.

Autoscaler TBM class

An object from this class, represents a TBM auto-scaler that is going to interact with the environment (with a Jitsi object).

This class will have 3 attributes:

- Jitsi: the environment to interact with
- Upper threshold
- Lower threshold

The most important method of this class will be:

- perform_action(state): given a state it will start a new jvb if mean CPU load is above the Upper threshold or it will stop a running jvb if mean CPU load is bellow Lower threshold. If it does one of these actions, then it waits 20 seconds of cooldown period ($20 > 15$ seconds needed to start or stop a jvb).

Note: for example if Upper is 60 and Lower is 30 we will call this TBM methods as "60-30 tbm" algorithm

Autoscaler RL class

An object from this class, represents the agent that is going to interact with the environment (with a Jitsi object).

This class will have 2 attributes:

- Jitsi: the environment
- Policy: the policy to follow

The most important method of this class will be:

- perform_action(state): given a state it will perform the action the policy is ordering for that state. If it does one of these actions (start or stop), then it waits 20 seconds of cooldown

period (20 > 15 seconds needed to start or stop a jvb).

The pseudo-code of the simulation is the following:

```
ROUND_COUNTER = 0   # Second of the cycle
TOTAL_ROUND_COUNTER = 0   # Second of the simulation
CYCLE_IN_SECONDS = 600   # 600 or 3600 seconds
TOTAL_PRICE = 0
PHOTO_INTERVAL = 5   # 5 seconds

jitsi = Jitsi()
autoscaler = None
populate_timetable_for_next_cycle()  # adding randomnes to the pattern...

if OPTION == 1:
    autoscaler = AutoscalerRL(policy)
else:
    autoscaler= AutoscalerTBM()

while TOTAL_ROUND_COUNTER < 604800: # 7 days
        state = jitsi.get_state()
        autoscaler.perform_action(state)
        advance_rounds(jitsi, PHOTO_INTERVAL) # polling strategy...
```

Table 12: Simulation code

```
def advance_rounds(jitsi,rounds)
    for (rounds):
        add_new_connections_for_this_round_to_jitsi(ROUND_COUNTER)
        update_price() # update money expended

        jitsi.advance_round()

        TOTAL_ROUND_COUNTER = TOTAL_ROUND_COUNTER + 1
        ROUND_COUNTER = ROUND_COUNTER + 1
        if ROUND_COUNTER == CYCLE_IN_SECONDS:
            ROUND_COUNTER = 0
            populate_timetable_for_next_cycle()  # adding randomnes to the pattern...
```

Table 13: advance_rounds() function

It is important to keep in mind that in Table 12 we are assuming that we already have a trained policy (Q learning has not been coded again) for the case of AutoscalerRL.

3.2 Quotas

We are going to keep track of the price paid after the 7 days of the simulation in order to be able to compare both algorithms. This price is going to be updated on each round (every simulated second) adding the price A of having a server running in the cloud provider for one second and adding the price B of unfulfilling the SLA agreement with our customer (using more than CPU_LOAD_SLA% resources of the system because it has been observed that the quality of the conference starts decreasing at this point) for one second:

```
def update_price(jitsi)
    TOTAL_PRICE = TOTAL_PRICE + A * jitsi.video_bridges_up
    if mean_cpu_load >= CPU_LOAD_SLA:
        TOTAL_PRICE = TOTAL_PRICE + B * (1 + (mean_cpu_load - CPU_LOAD_SLA)/CPU_LOAD_SLA)
```

Table 14: update_price() function

A and B have been set to real values that we can find in real scenarios. A has been set to 0.000025 (0.09€ per hour [23]) and B to 0.00025 (0.9€ per hour). So usually its much more expensive (x 10) the price that we have to pay to our users for unfulfilling SLA agreement than the price that we pay for renting the machines.

It should be noticed that the price that we pay to our users for unfulfilling the SLA agreement is not fixed, it will increase as we decrease the service quality.

3.3 Visualization

We have used a graphic library [24] in order to be able to visualize the traffic patterns we are fictitiously generating (in this case only with 2 video-bridges). This library is a helpful tool to design patterns.

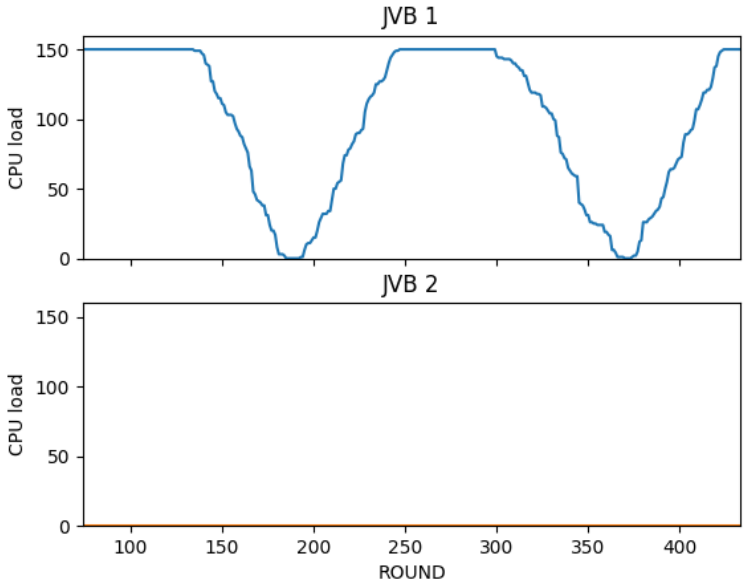

Figure 6: Simulation with auto-scaler disabled.

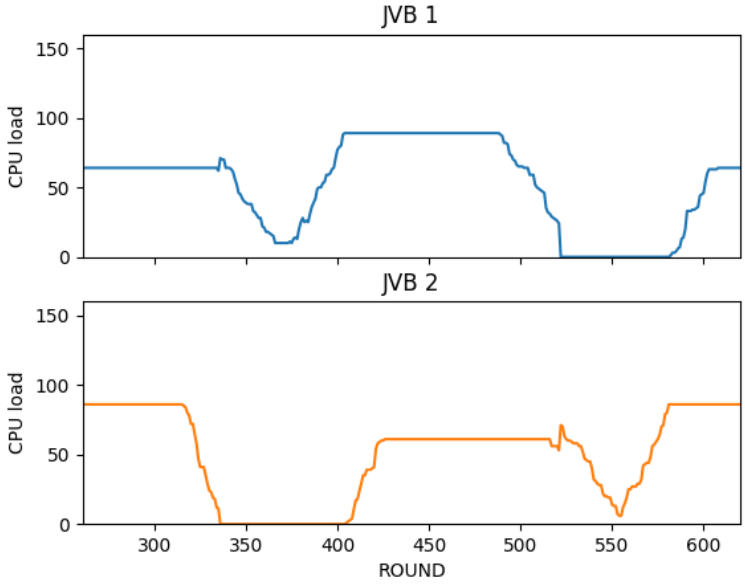

Figure 7: Simulation with TBM auto-scaler enabled.

In Figure 6 we see an example of how a single JVB reacts under a workload pattern that concentrates the conferences in a certain moment of the cycle. The auto-scaler is disabled so, the second JVB is not running.

In Figure 7 the same pattern is used but now a simple TBM is switching on and off the servers depending the workload the platform is holding.

4 Architecture of the solution

In this section we discuss the three design choices to be made: the state definition (how to represent the world), the action space (actions that the agent can do) and the reward function (the quality of each state).

4.1 Formulating the auto-scaling problem as an MDP problem

Let's formulate now our problem as and MDP problem. The first idea was to use this state definition:

$$(\#servers\ up,\ \#users\ connected,\ mean\ cpu\ load,\ second\ of\ the\ cycle) \tag{6}$$

Note that with this state definition, the state space has more than several thousand million states. Having in mind that our toy problem had only 49 states, we decided to decrease this state space to make it tractable. This is because in Q learning algorithm it is necessary to visit each of the states many times to obtain a good estimate of Q function. So we should train/interact with the environment too much time before having a good policy.

We decided to drop the users connected dimension making our agent less intelligent but easier to train. We also decided to reduce the dimension of mean CPU load and second of the cycle discretizing them by segments of 20 units.

So the state will be defined finally as follow:

$$(\#servers\ up,\ mean\ cpu\ load,\ second\ of\ the\ cycle) \tag{7}$$

Where the values that each dimension can take are the following ones:

- # servers up = {1,..,15}. With the traffic patterns we are going to generate it is enough.
- mean cpu load = {0,1,...,200}. We allow a server to go beyond its 100% load capacity. For example if the server is at its 95% capacity and we add 1 video-conference more, the load will raise to 105%. It has been observed empirically that the mean CPU load never exceed the 4000% in the different fictitious traffic patterns that we are going to use in the evaluation, thus the value 200 (4000/20).
- second of the cycle = {0,1,2...,29} or {0,1,2...,179} depending on the version (600/20 or 3600/20)

Now with this state definition the state space has 15 x 201 x 30 = 90450 states in first version and 15 x 201 x 180 = 542700 states in the second.

The three possible actions will be:

$$\{+1\ server,\ -1\ server,\ do\ nothing\} \tag{8}$$

So we have this number of possible policies: 3^{90450} and 3^{542700} respectively.

Note that not all actions can be performed from all states (for example, we should always have a minimum of one server up and a maximum of 15), so in fact the number of policies would be a little bit smaller.

One limitation that this video-conferencing system has is that it is not possible to reallocate one user (that has been already allocated in a server) into another server. If this was possible we would have specify this as possible actions.

The reward function will be defined as minus the price that you would pay in the next PHOTO INTERVAL seconds if anything changes (having in mind that PHOTO INTERVAL is the time between two photos of the system):

$$Reward(s) = -(\#server\ up * A * PHOTO\ INTERVAL + SLA) \tag{9}$$

Where SLA is just a function in the following format:

$$SLA = \begin{cases} (1 + \frac{(mcl - CPU\_LOAD\_SLA)}{CPU\_LOAD\_SLA}) * B * PHOTO\ INTERVAL & \text{if } mcl > CPU\_LOAD\_SLA \\ 0 & \text{else} \end{cases} \tag{10}$$

Where *mcl* stands for mean CPU load.

4.2 Other formulations of the state space

We could try to train our agent with a different formulation of the state. For example we could try:

$$(\#servers\ up, mean\ cpu\ load, mean\ cpu\ load\ t\ minus\ 1, mean\ cpu\ load\ t\ minus\ 2) \tag{11}$$

Where mean cpu load t minus 1 and mean cpu load t minus 2 are simply the mean CPU usage in the last two previous photos of the system (that is 10 and 5 seconds ago as PHOTO_INTERVAL will be always 5 in our characterization).

Note that this state space definition is independent of ROUND_COUNTER (the second of the cycle) so probably it would scale better to bigger cycles without the need of increasing the number of states.

This has not been implemented due to time restrictions.

5 Evaluation

As mentioned above, we are going to evaluate both algorithms with patterns that repeat themselves in cycles of 10 minutes (first version) and cycles of 1 hour (second version). Each of the experiments versions is going to be tested against two type of patterns.

The first version (10 minutes) will be tested with:

- A pattern where video-conferences initializations are uniformly distributed during the cycle.
- A pattern where 50% of the initialitzations are uniformly distributed in all the cycle but the other 50% uniformly distributed in a 50 second range space (so we will have a higher concentration in that range space).

The second version (1 hour) will be tested with:

- A pattern where video-conferences initializations are uniformly distributed during the cycle.
- A pattern where video-conferences initializations are uniformly distributed during the cycle but the load (number of conferences) will be higher.

This kind of patterns have been selected in order to have a variety of situations that help us to understand how the algorithms behave.

We have decided to take a PHOTO_INTERVAL of 5 seconds. It is worth to mention that if we increase this time window we are probably going to decrease the performance of TBM algorithm more than the performance of the RL (that is more robust against these changes). This is because the RL algorithm will simply learn to be more conservative in order to prevent reaching SLA levels. In contrast, TBM algorithm will not be able to react during the time window until the next PHOTO_INTERVAL decision, so it will be more time unfulfilling the SLA requirements.

It is important to notice that as we need usually more than several hundreds of thousands or even millions of interactions (in 1h cycle version that is the most realistic situation) with the simulator in order to get a good policy (and each interaction represents and action from the auto-scaler and 5 seconds of waiting time), it would be unfeasible to train it against the real environment as it would take at least 600 000 * 5 seconds $\approx$ 1 month.

Each of the experiments (with each configuration) has been executed 100 times and results averaged. This has been possible with SERT Computation Cluster of the Computer Architecture Department of UPC [25].

The mean duration of all the video-conferences is 120 seconds and they are all composed by five users of type 1 (audio and camera disabled) and one user of type 2 (camera and audio enabled). The CPU_LOAD_SLA has been set to 70%.

5.1 First version (cycles of 600 seconds)

First we test both algorithms against workload patterns of 10 minute cycles.

5.1.1 Pattern of connections 1 (uniformly distributed)

The number of conferences is 50 per cycle.

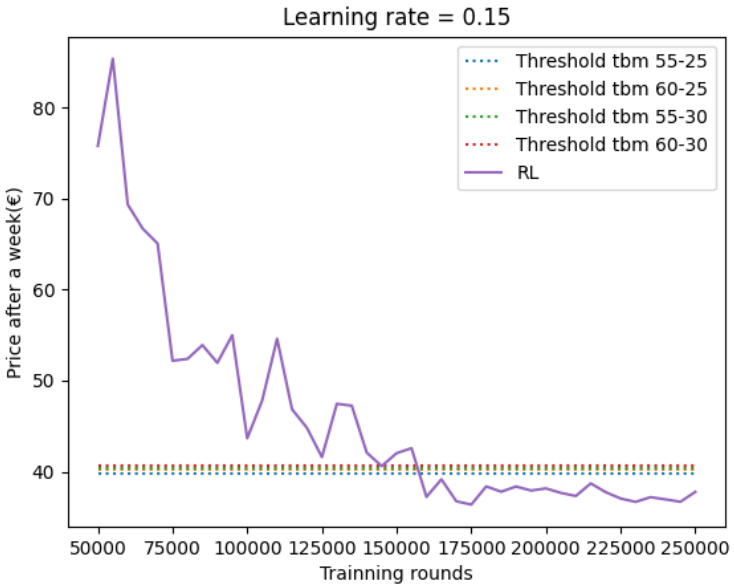

Figure 8: Results RL algorithm VS TBM methods (Pattern 1)

The 4 TBM methods used in the comparison are the best 4 that wee could found as defined here 3.1: The blue one is the $55 - 25$ tbm, the yellow one is the $60 - 25$ tbm, the green one the $55 - 30$ tbm and the red one the $60 - 30$ tbm.

We can observe in Figure 8 that after 200 000 iterations of training our agent is able to beat any of the possible TBM methods. After that it seems that is not worth to continue with the training.

Even if it seems that there is a little difference, this is the difference in price for one single week. For a complete year it would be multiplied by 52 (52 weeks in one year).

We could probably get better results if instead of discretizing the time dimension and cpu dimensions in units of 20 we do it in units of 10 because our agent will have a finer input of what is happenning. However if we discretize in smaller segments we will need longer trainning periods as we have more states to visit.

Also we could mention that the learning rate has been fixed to 0.15. We could try to decrease the learning rate from 0.15 to 0.05 in the middle of the training after 200 000 iterations and see if we get further improvements.

5.1.2 Pattern of connections 2 (concentration in a 50 second range space)

The number of conferences is 50 per cycle.

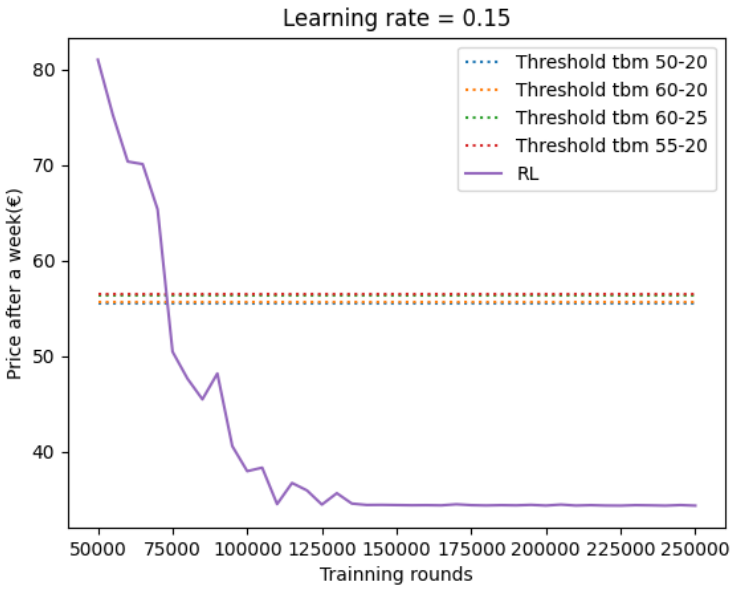

Figure 9: Results RL algorithm VS TBM methods (Pattern 2)

In Figure 9 we observe that RL is also able to do better than the best TBMs. Moreover it takes less rounds than in pattern 1.

We can see that RL algorithm is the best algorithm in the 2 different situations after sufficient rounds of training. It is important to notice that the gap in performance between the TBM methods and the RL algo increase as the pattern is more complex.

It is worth mentioning that we have only considered thresholds that are multiple of 5. We would probably get better results if finer thresholds were considered for the TBM.

5.2 Second version (cycles of 3600 seconds)

Next we test both algorithms against workload patterns of 1 hour cycles. This is a more realistic scenario in applications.

5.2.1 Pattern of connections 1 (uniformly distributed)

The number of conferences is 300 per cycle. This decision is because we have multiplied the number of conferences by 6 (as the cycle length has been multiplied by 6) and thus be able to compare both results (pattern 1 for version 1 and 2).

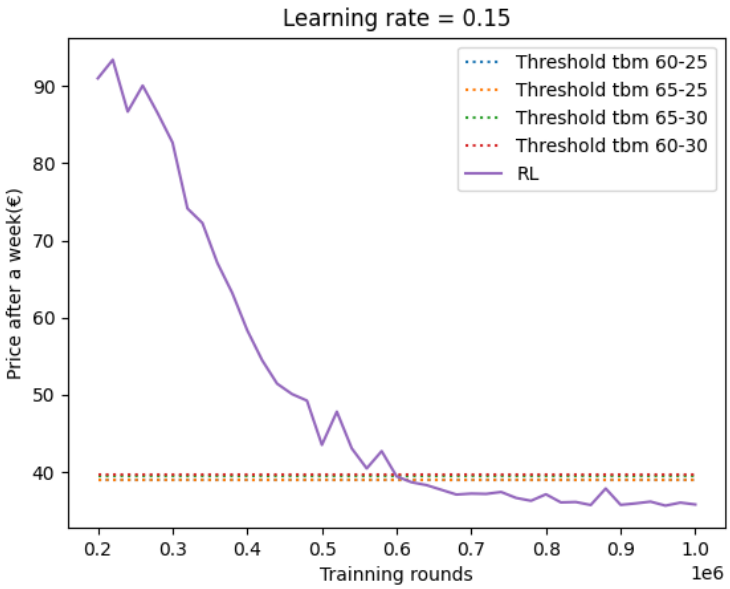

Figure 10: Results RL algorithm VS TBM methods (Pattern 1)

As we can observe in this Figure 10, as the state space gets bigger (because the cycle is bigger) we need more and more interactions with the environment to obtain a good policy. We can have an idea of how many iterations would be necessary to train a RL algorithm if cycles were of 24 hours.

Here we can see that we would need more than 600 000 iterations to perform better than TBM algorithms. After that it seems that is not worth to continue with the training.

5.2.2 Pattern of connections 2 (higher load)

The number of conferences is 900 per cycle.

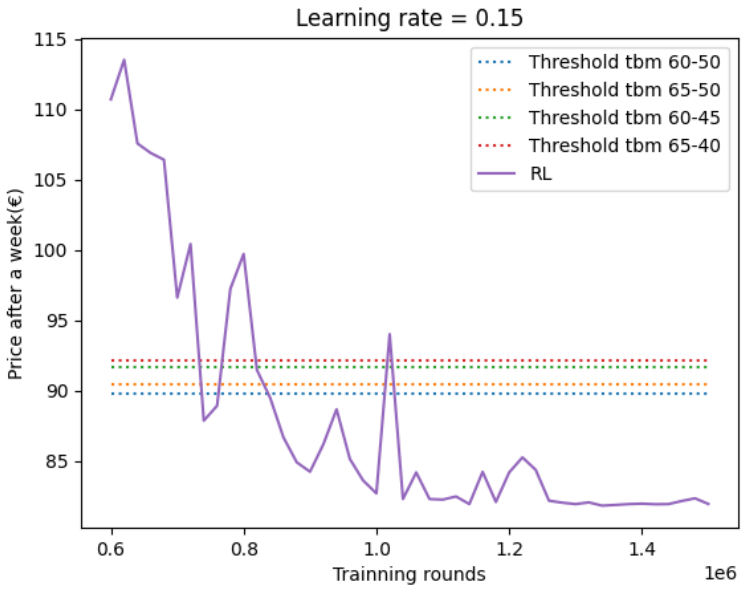

Figure 11: Results RL algorithm VS TBM methods (Pattern 2)

In this figure 11, as before, the difference in performance between the algorithms increase as the complexity (higher load) of the pattern increase (the difference is almost 10 € summing renting and SLA violation costs, in contrast to ≈ 5€ in pattern 1).

Even that 100 experiments were done with each configuration we can see that there exist high variance in the results, so we do not see a smooth line. Nevertheless this variance decreases as the number of iterations increase.

6 Conclusions

In this work we have develop a auto-scaling algorithm based on Reinforcement learning and analysed how it performs against different workload patterns in comparison with threshold based methods. The analysis is done with a simulator that imitates a real video-conference platform such as Jitsi.

6.1 Concluding remarks

The first conclusion that we could get from this work, is that although TBM algorithms are easier to deploy, as they do not need any training, RL algorithms achieve better results determining the optimum number of servers in a distributed application. Also, the difference is accentuated as more complex the pattern is as we saw with pattern 2 in each of the versions.

Moreover, if we implement online learning, RL will be able to adapt to the new situation if there are smooth changes in load patterns and these persist over time (in contrast to TBM algorithms that will downgrade its performance as they are set for a particular load pattern).

It is true however, that RL will not be able to adapt to rapid changes that do not last enough time to learn them (as it needs many interactions with the same transition probabilities to get a good Q estimation).

The workflow with any problem using RL could be like this: obtaining a first Q function and policy through interacting with a simulator and then getting a more accurate solution with online training against the real environment (with the Q function and policy obtained with the simulator as the starting point). Starting the training directly on the real scenario without using a simulator implies prohibitive amount of time as an important number of iterations are needed before obtaining a good policy (better than TBM).

Although it has not been stated in section 5 (Evaluation), we can conclude as well that it is a little bit tricky to tune the parameters in order to achieve an effective learning. The learning rate α should not be to small (or we will get trapped in local optimums) neither too big (we will oscillate and not converge). Another tricky tuning parameter is ϵ (the exploration parameter). If we decay ϵ too slow we will explore too much and not exploit to the promising direction (as we have a limited amount of interactions). If we decay ϵ to fast , very few exploration will be made and then probably we are going to end in a local optimum.

Also the definition of the reward function is vital and a key element to help our agent take appropriate decisions and other definitions could be investigated. The designer could try with different reward functions and eventually pick the one that makes the agent learn faster or a better policy.

6.2 Future work

6.2.1 Online learning

In our scenario we supposed that the dynamics of the environment are not changing through time (same connection patterns from users through time). Nevertheless if it is the case that connection patterns change, it would be interesting to keep learning and updating our Q function (and thus the policy) to adapt to the new situation. This is actually how most of Reinforcement Learning solutions are implemented.

6.2.2 Comparison of algorithms on the real scenario

In order to do the comparison on the real scenario, Jitsi servers should be deployed in physical or virtual machines following the official guides. The generation of load by clients would be automated using scripts. In Annex 1 we have described the first steps to prepare the scenario.

6.2.3 Deep Q learning

When the state space is too big, it is infeasible to visit all state action pairs in order to have an accurate estimate of the $Q(s, a)$ function so Q-learning algorithm doesn't work. An alternative way to deal with this problem, is to use deep reinforcement learning, i.e., instead of representing $Q(s, a)$ as a table, it is represented as a deep neural network (state and action as inputs and Q value as output) that is able to generalize to state action pairs not seen before. With this idea we could make the state-space richer, including weekday, the number of connected users (or including more old CPUs usages in formulation 4.2). Also discretizing in smaller units or no discretizing at all. And thus we could see if we get better results (in terms of money expended or in terms of number of interactions needed until a good policy is obtained).

Annex 1: Preparing a real scenario to test the obtained policy

The comparison of the auto-scalers (TBM vs RL) in the real scenario has not been implemented. Nevertheless the first steps preparing the scenario have been documented.

Generating fictitious workload patterns

In order to create fictitious load we have deployed a cluster of video-conference clients (web-browsers) ready to connect to the server. To automate the process of connecting to the server we used the Selenium library. Selenium is a Python library that allow us to interact with a browser [26].

Chrome, the browser chosen in our case, has been used in its headless mode to allow machines without a graphic environment to execute it. To emulate real users connecting to the server we will provide a fake video and fake audio to our clients to use it as if it were the webcam and micro output (a feature offered by Chrome).

The cluster of clients have been deployed in UPC Cloud. We have rented machines of 1 core and 1 GB of memory. Each of this machines runs 5 clients (5 browsers) that simulates users connected to Jitsi with camera off and audio muted. We have also rented machines of 4 cores and 2 GB of memory. Each of this machines runs 1 client that simulates a user connected to Jitsi with camera and audio on.

This configuration of clients pretend to simulate common scenarios for example in online teaching where the teacher has the camera an audio on but students do not.

An important problem that we have found during this work is the huge amount of CPU resources needed in order to stress a single JVB. We need a lot of users connected to do so and to simulate a user a browser is needed,which is computationally expensive when dealing with video and audio because of the encoding a decoding process.

Alternatives have been considered, as building our own light WebRTC client with open source libraries, but this has been discarded because the limited amount of time. Another alternative could be to use services offered by companies like WebRTC Test [27] that offer doing massive tests against your WebRTC servers, but this has also been discarded due to price constraints. Consequently limiting the resources of the VM hosting the JVBs (to be able to stress them with less requests) has been the chosen approach.

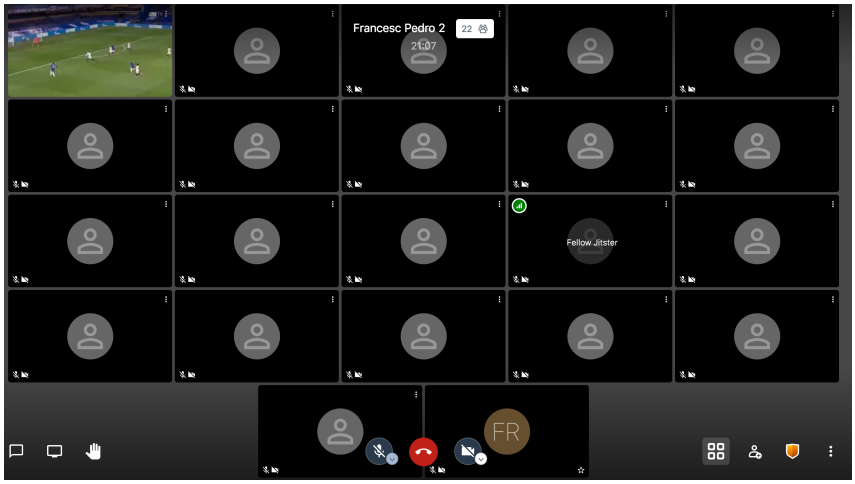

Figure 12: Example of stress test with a football fake video as camera input.

Monitoring the system

The Jitsi servers will be deployed in VMs in our own physical servers. The infrastructure will consist in 3 VMs, one hosting the Web server, Prosody, Jicofo and one JVB, and the other 2 just hosting a JVB. We will always start our experiments with just one JVB up.

The capacities of each VM are as follows:

- 0.5 CPUs (limited by the hypervisor to ease the process of stressing the machine)
- 0.5 GB RAM

To monitor how the system behaves we have used a software suite composed of Prometheus and Graphana. Prometheus is a time series database that is going to collect all the output metrics of Jitsi. Graphana will be used to visualize this metrics with graphs that are going to help us understand better the situation.

Figure 13: Example of Graphana output under two different loads.

We would use the HTTP API of prometheus [28] to monitor the system and get the CPU usage of each machine. This will API will be called each 5 seconds (polling strategy).

It will not be necessary to call any hypervisor API to boot machines as we would already have the machines running (with jitsi service running or not running). This is unrealistic and it beats the purpose of this work (because we would be paying renting costs always), but it will simplify the test. We could just use an script with a delay that eventually starts the JVB intead of starting the JVB directly (to emulate real booting processes).

In the future we could consider also to deploy the Jitsi servers on Docker containers instead of VMs as it is know that the booting process is faster.

Aditional considerations

It could also be interesting, in future versions of the algorithm, to investigate Jicofo log files (where there is information about new connections of users, disconnections, JVB events, etc.) as they could provide useful information. Using a simple XMPP client to connect to Prosody to listen to rooms where JVBs and Jicofo talk, could also be insightful.

References

[1] Video-conference requeriments, *https://www.freeconference.com/blog/the-minimum-speed-required-for-video-conferencing/*

[2] Guifi Common Network Website, *https://guifi.net/es*

[3] Jitsi Website Project, *https://jitsi.org/*

[4] Jitis Auto-scaler, *https://github.com/jitsi/jitsi-autoscaler*

[5] Chistera, Leading Edge, *https://www.chistera.eu/projects/leadingedge*

[6] Tania Lorido-Botran · Jose Miguel-Alonso · Jose A. Lozano, *A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments*, Journal of Grid Computing manuscript

[7] AWS Auto-scaling, *https://aws.amazon.com/es/autoscaling/*

[8] Flexera RightScale Docs, *https://docs.rightscale.com/*

[9] Ying Liu · Navaneeth Rameshan · Enric Monte, *ProRenaTa: Proactive and Reactive Tuning to Scale a Distributed Storage System*, 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

[10] Daniel Jacobson · Danny Yuan · Neeraj Joshi, *Scryer: Netflix's Predictive Auto Scaling Engine*, The Netflix tech blog

[11] Daniel Jacobson · Danny Yuan · Neeraj Joshi, *Scryer: Netflix's Predictive Auto Scaling Engine — Part 2*, The Netflix tech blog

[12] Yisel Gari · David A. Monge · Elina Pacini, *Reinforcement Learning-based Application Autoscaling in the Cloud: A Survey*, Engineering Applications of Artificial Intelligence

[13] Xavier Dutreilh · Sergey Kirgizov · Olga Melekhova, *Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: Towards a Fully Automated Workflow*, ICAS 2011 : The Seventh International Conference on Autonomic and Autonomous Systems

[14] Christopher Watkins · Peter Dayan, *Q-learning*, Machine Learning

[15] Naghmeh Dezhabad · Saeed Sharifian, *Learning-based dynamic scalable load-balanced firewall as a service in network function-virtualized cloud computing environments*, The Journal of Supercom- puting, 2018

[16] Enda Barrett · Enda Howley · Jim Duggan, *Applying reinforcement learning towards automating resource allocation and application scalability in the cloud*, Concurrency Computation Practice and Experience, 2012

[17] Pooyan Jamshidi · Aakash Ahmad · Claus Pahl, *Autonomic Resource Provisioning for Cloud-Based Software*, SEAMS'14

[18] Hamid Arabnejad · Claus Pahl · Pooyan Jamshidi, *A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling*, 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

[19] SFU vs MCU, *https://quobis.com/es/2021/05/03/sfu-vs-mcu-cual-es-la-mejor-forma-de-gestionar-una-multiconferencia/*

[20] WebRTC Website Project, *https://webrtc.org/*

[21] Repository with the code of the simulator and algorithms, *https://gitlab.com/francescroy/rl_real_case_2*

[22] Grafana software, *https://grafana.com/*

[23] Amazon Web Services pricing of virtual machines, *https://aws.amazon.com/ec2/pricing/on-demand/*

[24] Python graphic library, *https://matplotlib.org/*

[25] DAC Services, *https://www.ac.upc.edu/ca/nosaltres/serveis-tic/servicios-1*

[26] Selenium's framework web, *https://www.selenium.dev/*

[27] WebRTC Test, *https://testrtc.com/*

[28] HTTP Prometheus API guide, *https://prometheus.io/docs/prometheus/latest/querying/api/*