# Rapport sur :

# Contrôle optimal des arrivées à un système de file d'attente avec panne

Realisée par :
SOUKI SALMA
ZINEB BARAKA
MERIEM EL AMARI

# Partie Théorique :

## Description et formulation du problème :

Nous considérons un système de file d'attente composé de deux files d'attente parallèles, dans lequel :

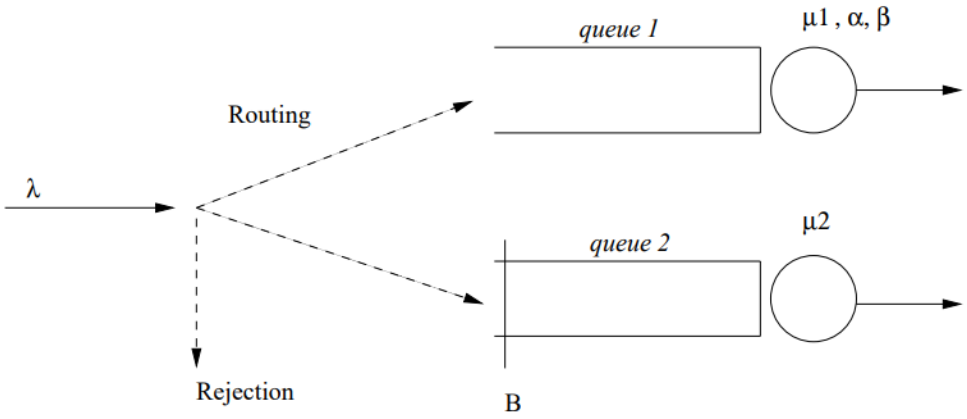
• Les paquets arrivent au système dans un flux de Poisson avec un débit A,

• Les paquets sont servis dans les files d'attente 1 et 2 par deux serveurs exponentiels à des débits et P2 respectivement,

• Le premier serveur est sujet à des pannes intermittentes,

• Les temps de panne et de réparation suivent deux distributions exponentielles aux taux a et o respectivement,

• La file d'attente 2 est la capacité finie B




Un paquet arrivant sera soit rejeté, soit envoyé à l'une des deux files d'attente.

- $\xi$ : Coût de rejet.
- $((x_t^1, x_t^2), s_t)_{t>0}$: L'état du système à l'instant t, avec des valeurs dans S = IN × {0, 1, ..., B} × {0,1}, où $x_t^i$ est le nombre total de paquets dans la file d'attente i (y compris le paquet dans le serveur) (i = 1,2).
- $s_t$: L'état du premier serveur, où

$$s_t \begin{cases} 0 & Si\ le\ serveur\ est\ en\ panne \\ 1 & sinon \end{cases}$$

- $z_t(x,s)$: Une action admissible dans l'état (x, s) à l'instant t, avec des valeurs dans D= {0, 1, 2}, où :

$$Z_t(x,s) = \begin{cases} 0 & Si\ un\ paquet\ arrivant\ est\ refusée \\ 1 & Si \text{ un paqet arrivant est acheminé vers la file d'attente 1} \\ 2 & Si \text{ un paqet arrivant est acheminé vers la file d'attente 2} \end{cases}$$

Et $x = (x^1, x^2)$ l'état de deux files d'attente.

- $(z_t\ , t > 0)$: une Stratégie de Contrôle (CS) admissible, à valeur dans A = DS.
- P : l'ensemble de toutes les stratégies de contrôle admissibles.

$$lim\,supE\pi(\int_0^T \exp(-\delta t)\,(\sum_{i=1}^{2} c_i\, x_t^i + \lambda.\xi\,)dt)$$

Une politique admissible qui est un processus stochastique indépendant identiquement distribué et pour lequel la distribution commune a toute sa masse concentrée en un point de A, est appelé un SC stationnaire.

Comme les temps inter-arrivées et inter-départs des paquets sont exponentiellement distribués et que l'ensemble d'action admissible D est fini, alors un CS optimal existe et il est stationnaire (Lippman et Walrand).

Ainsi, nous restreignons l'attention aux CS stationnaires et définissons le coût d-actualisé à partir de l'état initial (x,s) associé au problème (P1) par :

$$J^{\delta}(x,s) = \min_{z \in P_S} E^{z}_{(x,s)}[\int_0^{\infty} e^{-\delta.t}(\sum_{i=1}^{2} c_i x_t^i + \lambda.\xi \mathbb{1}_{\{z_t=0\}})dt]$$

où Ps CP désigne l'ensemble de tous les politique stationnaires.

Soit : to , t1 <..... < tn ... les instants dans le temps dénotant les époques de transition de l'état du système ((t, St), t > 0).

Soit :

$$V_n^{\delta}((x,s),z) = E_{(x,s)}^{z}[\int_0^{t_n} e^{-\delta t}(\sum_{i=1}^{2} c_i x_t^i + \lambda.\xi \mathbb{1}_{\{z_t=0\}})dt]$$

Et

$$J_n^{\delta}(x,s) = \min_{z \in P_S} V_n^{\delta}((x,s),z), \quad n = 0,1,...$$

Avec :

$$J^{\delta}(x,s) = \lim_{n\to\infty} J_n^{\delta}(x,s)$$

## Equation équivalente dans le cas discret :

$$Pr[t_{k+1} - t_k > t] = e^{-t(\lambda+\mu_1+\mu_2+\alpha+\beta)}$$

On suppose :

$$\Phi = \frac{\lambda+\mu_1+\mu_2+\alpha+\beta}{\lambda+\mu_1+\mu_2+\alpha+\beta+\delta}$$

Donc :

$$V_n^{\delta}((x,s),z) = E_{(x,s)}^{z}[\int_0^{t_n} e^{-\delta t}(\sum_{i=1}^{2} c_i x_t^i + \lambda.\xi \mathbb{1}_{\{z_t=0\}})dt]$$

Et puis :

$$\widetilde{V}_n^{\Phi}((x,s),z) \triangleq E_{(x,s)}^{z}[\sum_{k=0}^{n-1} \Phi^k(\sum_{i=1}^{2} c_i x_k^i + \lambda.\xi \mathbb{1}_{\{z_k=0\}}]$$

Définissez le coût minimum actualisé pour les systèmes à temps discret à n étapes et à horizon infini, respectivement, par :

$$\widetilde{J}_n^{\Phi}(x,s) \triangleq \min_{z \in P_S} \widetilde{V}_n^{\Phi}((x,s),z)$$

Et :

$$\widetilde{J}^{\Phi}(x,s) \triangleq \min_{z \in P_S} \widetilde{V}^{\Phi}((x,s),z)$$

Comme précédemment, On peut démontrer aussi que :

$$\widetilde{J}^{\Phi}(x,s) = \lim_{n\to\infty} \widetilde{J}_n^{\Phi}(x,s)$$

Pour chaque état initial $(x, s)$ , *on a que* :

$$V_n^{\delta}(x,z) = \frac{1-\Phi}{\delta}\widetilde{V}_n^{\Phi}((x,s),z)$$

L'équivalence, au sens du coût actualisé optimal, entre (P1) et la formulation en temps discret découle aisément de (2), (5) et (6) par

$$J^{\delta}(x,s) = \frac{1-\Phi}{\delta}\widetilde{J}^{\Phi}(x,s)$$

D'après le résultat de Walrand, Jo (r, s) est l'unique solution bornée du EPD suivant :

$$\widetilde{J}^{\Phi}(x,s_1) = \min_{z\in P_S}\{c(x,s_1;z) + \Phi\sum_{y,s_2} P(y,s_2|x,s_1;z)\widetilde{J}^{\Phi}(y,s_2)\}$$

c(...) est la fonction de coût instantanée,

P((y, S2)|(T, S1), 2) est la probabilité conditionnelle (pour le temps discret

problème) que le système passe à l'état (y, s2) à l'instant n + 1 lorsque l'action z(1, 81) lui est appliquée à l'instant n.

$$\begin{aligned}
T\widetilde{J}^{\Phi}(x,0) &= cx + (\delta+\gamma)^{-1}.\Phi\{\mu_1\widetilde{J}^{\Phi}(x,0) + \mu_2\widetilde{J}^{\Phi}(D_2x,s) \\
&+ \alpha\widetilde{J}^{\Phi}(x,0) + \beta\widetilde{J}^{\Phi}(x,1) \\
&+ \lambda.min\{\widetilde{J}^{\Phi}(x,0)+\xi, \widetilde{J}^{\Phi}(A_1x,0), \widetilde{J}^{\Phi}(A_2x,0)\mathbb{1}_{\{x_2<B\}}\}
\end{aligned}$$

$$\begin{aligned}
T\widetilde{J}^{\Phi}(x,1) &= cx + (\delta+\gamma)^{-1}.\Phi\{\mu_1\widetilde{J}^{\Phi}(D_1x,1) + \mu_2\widetilde{J}^{\Phi}(D_2x,1) \\
&+ \alpha\widetilde{J}^{\Phi}(x,0) + \beta\widetilde{J}^{\Phi}(x,1) \\
&+ \lambda.min\{\widetilde{J}^{\Phi}(x,1)+\xi, \widetilde{J}^{\Phi}(A_1x,1), \widetilde{J}^{\Phi}(A_2x,1)\mathbb{1}_{\{x_2<B\}}\}
\end{aligned}$$

Avec :

$$: x = (x_1,x_2) \in \mathbb{N}\times\{0,1,...,B\}, \quad cx = \sum_{i=1}^{2} c_i x_i, \quad \gamma = \lambda+\mu_1+\mu_2+\alpha+\beta,$$

$$D_1x = ((x_1-1)^+, x_2), \quad D_2x = (x_1, (x_2-1)^+), \quad x^+ = max(0,x),$$
$$A_1x = (x_1+1, x_2), \quad A_2x = (x_1, min(x_2+1, B)).$$

# Partie pratique :

## Politique d'itération :

### Définition :

La politique d'itération est un algorithme itératif pour résoudre des problèmes de décision de Markov en trouvant la politique optimale qui maximise la récompense cumulative. Il fonctionne en évaluant la politique courante et en l'améliorant en choisissant la meilleure action dans chaque état.

### Implémentation :

Il y a 3 fonctions principales dans notre code : "policy_evaluation_improvement" et "policy_iteration" et la fonction initial P .

Voici une explication de leur fonctionnement :

```
def policy_evaluation_improvement(policy, S, working,
lamda=lamda):
    V = {s: 0 for s in S}
    x = 0
    while True:
        x = x + 1
        oldV = V.copy()
        newPolicy = {}
        for s in S:
            a = policy[s]
            # V[s]=R(s,a)+sum(P(s_next,s,a)*oldV[s_next]
for s_next in S)
            sClientAjouterFile1 = (min(s[0] + 1, N1),
s[1], s[2])
            sClientAjouterFile2 = (s[0], min(s[1] + 1,
N2), s[2])

            if s[2] == 0:
                min1 = min(oldV[s] + xi,
oldV[sClientAjouterFile2])
            elif s[1] == N2:
                min1 = min(oldV[s] + xi,
oldV[sClientAjouterFile1])
            elif s[1] == N1:
                min1 = min(oldV[s] + xi,
oldV[sClientAjouterFile1])
            else:
                min1 = min(V[s] + xi,
V[sClientAjouterFile1], V[sClientAjouterFile2])
```

```
            if min1 == V[s] + xi:
                indexOfAction = 2
            elif min1 == V[sClientAjouterFile1]:
                indexOfAction = 0
            elif min1 == V[sClientAjouterFile2]:
                indexOfAction = 1
            newPolicy[s] = indexOfAction
            a = newPolicy[s]
            V[s] = cout(s, indexOfAction) + pow(delta +
gamma, -1) * phi * (
                    sum(P(s_next, s, a, working) *
oldV[s_next] for s_next in S) + lamda * min1)
        if all(abs(oldV[s] - V[s]) < 0.1 for s in S):
            return [V, newPolicy]
```

- *"policy_evaluation_improvement(policy, S, working, lamda=lamda)"* : Cette fonction prend en entrée une politique "policy", un ensemble d'états "S", un booléen "working" qui indique si le système fonctionne normalement ou non, et un paramètre "lamda" qui est une constante. Elle retourne une liste de deux éléments : le dictionnaire des valeurs de chaque état "V" et le dictionnaire de la nouvelle politique mise à jour "newPolicy".

  La fonction utilise l'algorithme d'évaluation de politique itérative pour estimer la valeur de chaque état dans l'environnement. Elle commence par initialiser les valeurs de chaque état à zéro. Elle itère ensuite jusqu'à ce que la différence entre les anciennes et les nouvelles valeurs de chaque état soit inférieure à un seuil fixé. À chaque itération, elle calcule la valeur de chaque état en fonction de sa politique actuelle, en prenant en compte les récompenses associées à chaque action, les probabilités de transition entre les états, et les valeurs des états suivants. Enfin, elle retourne les nouvelles valeurs de chaque état ainsi que la nouvelle politique optimale.

```
def policy_iteration(S, A, working=True, policy=policy):
    while True:
        old_policy = policy.copy()
        V = policy_evaluation_improvement(policy, S,
working)[0]
        policy = policy_evaluation_improvement(policy, S,
working)[1]
        print(policy_evaluation_improvement(policy, S,
working)[1])
        print("\n")
        if all(old_policy[s] == policy[s] for s in S):
```

```
            break
    return policy
```

- *"policy_iteration(S, A, working=True, policy=policy)"* : Cette fonction prend en entrée un ensemble d'états "S", un ensemble d'actions "A", un booléen "working" qui indique si le système fonctionne normalement ou non, et un dictionnaire "policy" qui représente la politique initiale. Elle retourne la nouvelle politique optimale mise à jour.

  La fonction utilise l'algorithme d'itération de politique pour trouver la politique optimale dans l'environnement. Elle commence par initialiser la politique à celle donnée en entrée. Elle effectue ensuite une évaluation de politique en utilisant la fonction "policy_evaluation_improvement" pour obtenir la valeur optimale de chaque état et la nouvelle politique associée. Elle répète ce processus jusqu'à ce que la politique ne change plus. Enfin, elle retourne la nouvelle politique optimale.

- La fonction *P (s_next, s, a, working)* calcule la probabilité de transition de l'état s à l'état s_next, sachant que l'action a est choisie et l'état de la première file.

```
def P(s_next, s, a, working):
    if working:
        if s_next[0] == s[0] and s_next[1] == s[1] and
s_next[2] == 0:
            # print("meme etat")
            return alpha
        elif s_next[0] == s[0] and s_next[1] == s[1] and
s_next[2] == 1:
            # print("changement de fonctionement")
            return beta
        elif s_next[1] == s[1] and (s_next[0] == s[0] - 1
or s_next[0] == 0) and s_next[2] == s[2]:
            # print("service file1")
            return mu1
        elif s_next[0] == s[0] and (s_next[1] == s[1] - 1
or s_next[1] == 0) and s_next[2] == s[2]:
            # print("service file2")
            return mu2
        else:
            # print("rien")
            return 0
    else:
        if s_next[0] == s[0] and s_next[1] == s[1] and
s_next[2] == s[2]:
```

```
            # print("meme etat")
            return alpha + mu1
        elif s_next[0] == s[0] and s_next[1] == s[1] and
s_next[2] != s[2]:
            # print("changement de fonctionement")
            return beta
        elif s_next[0] == s[0] and (s_next[1] == s[1] - 1
or s_next[1] == 0) and s_next[2] == s[2]:
            # print("service file2")
            return mu2
        else:
            # print("rien")
            return 0
```

La fonction renvoie un nombre qui représente la probabilité de transition de l'état s à l'état s_next. Cette probabilité dépend du mode de fonctionnement, de l'état courant s, de l'état suivant s_next et de l'action choisie a .

Plus précisément, si le système est en mode de fonctionnement, la fonction vérifie les conditions suivantes :

Si l'état suivant s_next est le même que l'état courant s et qu'aucun client n'arrive ou ne part de l'un des deux files, la fonction renvoie la probabilité α.

Si l'état suivant s_next est le même que l'état courant s, mais le système passe du mode de fonctionnement au mode de panne ou inversement, la fonction renvoie la probabilité β.

Si l'état suivant s_next est obtenu en ajoutant un client à la file 1 et que le nombre de clients dans la file 1 est inférieur à N1, la fonction renvoie la probabilité de service dans la file 1 μ1.

Si l'état suivant s_next est obtenu en ajoutant un client à la file 2 et que le nombre de clients dans la file 2 est inférieur à N2, la fonction renvoie la probabilité de service dans la file 2 μ2.

Sinon, la fonction renvoie 0.

Si le système est en mode de panne, la fonction vérifie les conditions suivantes :

Si l'état suivant s_next est le même que l'état courant s et qu'un client arrive dans l'un des deux files, la fonction renvoie la probabilité α plus la probabilité de service dans la file 1 μ1.

Si l'état suivant s_next est le même que l'état courant s, mais le système passe du mode de fonctionnement au mode de panne ou inversement, la fonction renvoie la probabilité β.

Si l'état suivant s_next est obtenu en ajoutant un client à la file 2 et que le nombre de clients dans la file 2 est inférieur à N2, la fonction renvoie la probabilité de service dans la file 2 μ2.

Sinon, la fonction renvoie 0.

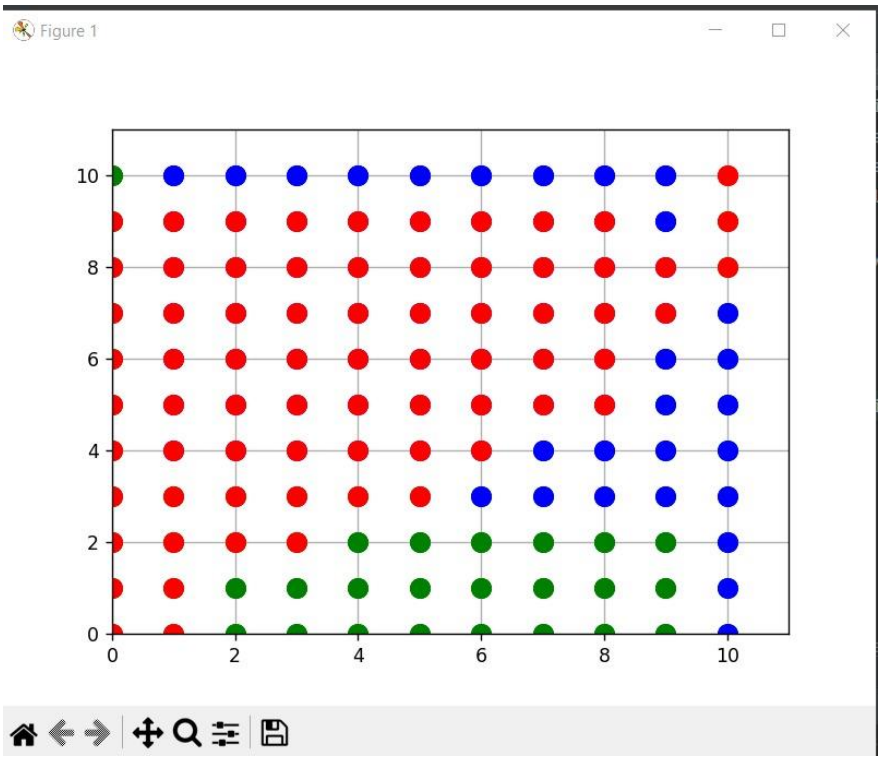
## ***Les résultats :***

Vert : zone d'acceptation dans la file 2 .

Rouge : zone d'acceptation dans la file 1

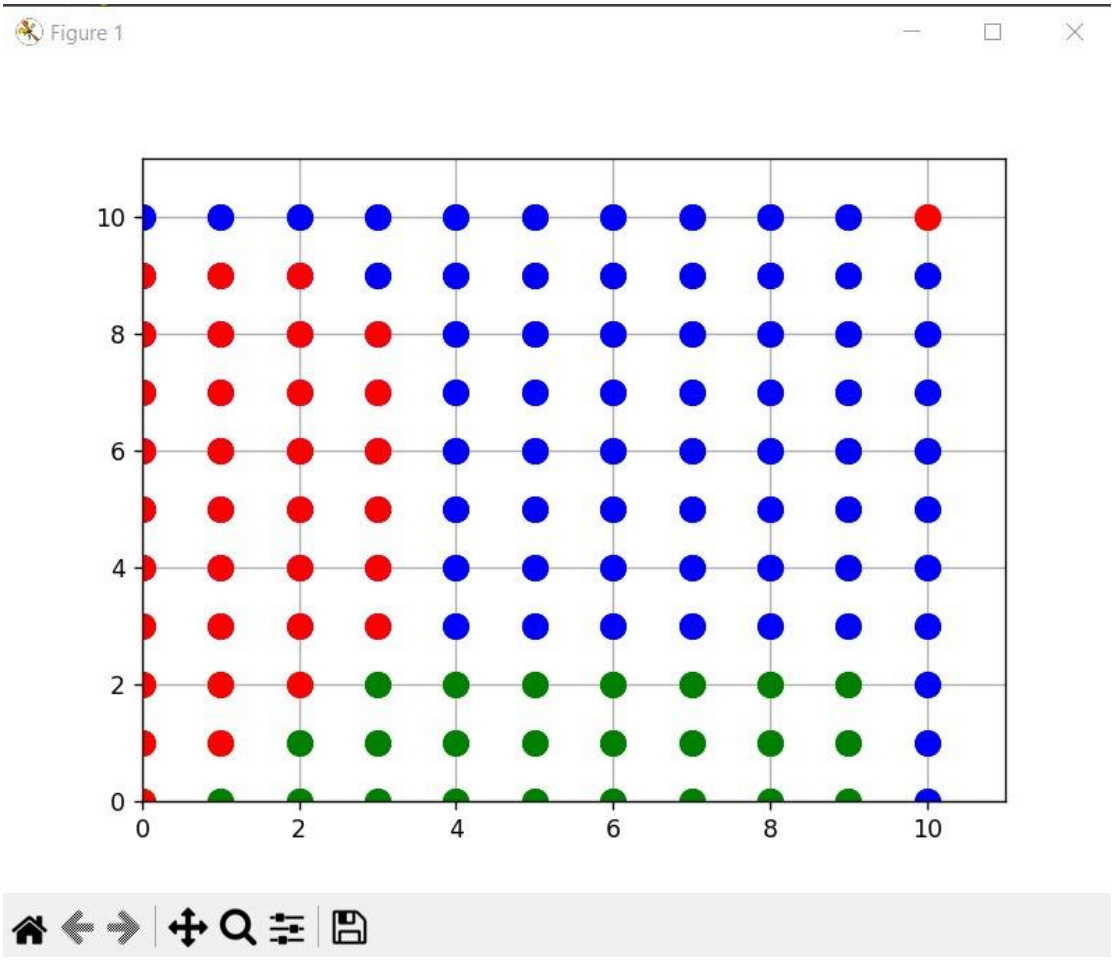
Blue : zone de rejet.

Pour les paramètres : c1=1, c2=5,$\xi = 90$ et la serveur 1 fonctionne.

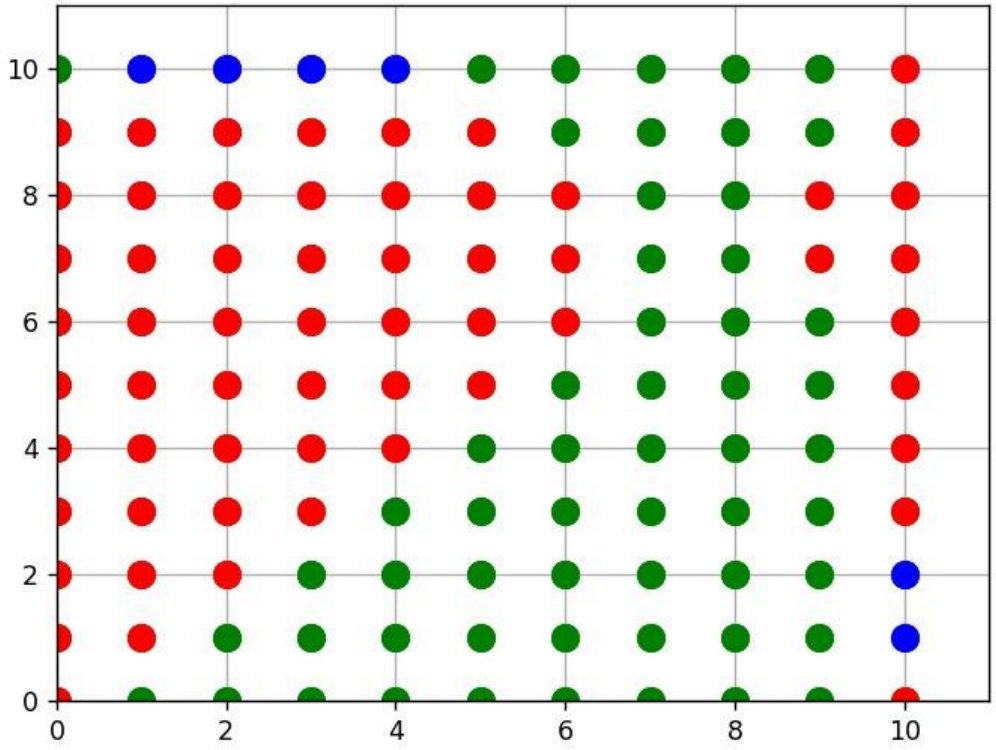



On remarque que la zone d'acceptation dans la file 1 plus grande que celle de la zone 2

Pour les paramètres : c1=4, c2=5,$\xi = 80$ le premier serveur fonctionne.



Pour les paramètres : c1=4 ,c2=5 ,$\xi = 300$




On remarque que la zone de rejet devient plus grande lorsque le cout d'acceptation dans la file 1 s'augmente

Pour les paramètres : c1=4, c2=5,$\xi = 80$ le serveur 1 ne fonctionne pas