

MODÉLISATION INCERTITUDE ET SIMULATION

RAPPORT DE PROJET 2023-2024



**ARIS
MEKSAOUI**



Table des matières

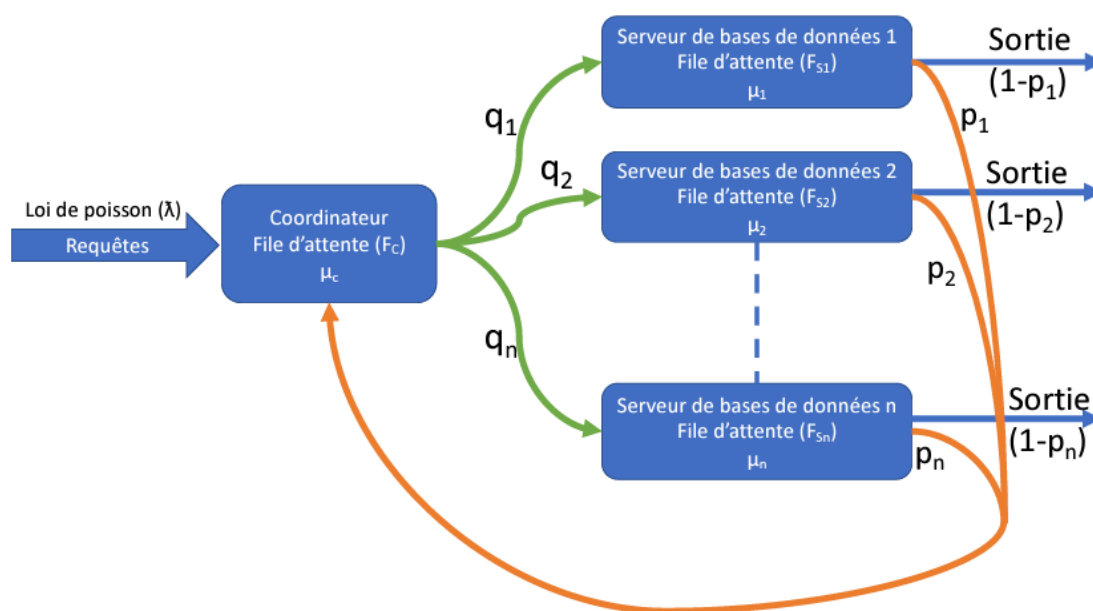
- **Rappel du sujet :**
- **Objectifs**
- **Comment procéder ?**
- **Simulation**
 - **Classes**
 - **File d'attente**
 - **MED**
 - **Coordinateur**
 - **Serveur i**
 - **Requete**
 - **Generateur de requetes**
 - **BDD:**
 - **Test :**
 - **Interface graphique :**
 - **Tests et Résultats**
- **Etude Analytique**
 - **Application numérique :**
- **Conclusion**

Modélisation Incertitude et Simulation

Rapport du projet

Rappel du sujet :

Dans le cadre de ce projet, nous explorons le fonctionnement d'une base de données distribuée sur plusieurs serveurs. Les requêtes des utilisateurs, générées selon une loi de Poisson, sont initialement dirigées vers un coordinateur. Ce dernier a pour mission de répartir ces requêtes de manière stratégique parmi les différents serveurs, en suivant des probabilités déterminées (q_i). Chaque serveur dispose d'un temps de traitement spécifique, caractérisé par une distribution exponentielle (μ_i). À la conclusion du traitement dans un serveur donné, la requête a deux options : elle peut quitter le système avec une probabilité (p_i) ou retourner au coordinateur pour être redirigée ultérieurement. L'étude approfondie de ce système s'articule autour de divers paramètres tels que le nombre de serveurs, les temps de traitement de ces serveurs, les taux d'arrivée des requêtes, ainsi que les probabilités de sortie du système.



Objectifs :

L'objectif central du projet est de mesurer la stabilité du système dans un état permanent. Cette évaluation repose sur une analyse approfondie des paramètres clés qui influent sur le fonctionnement du système et à identifier les configurations optimales et à déterminer les conditions sous lesquelles le système atteint un régime permanent stable.

Comment procéder ?

La méthodologie adoptée pour atteindre les objectifs du projet repose sur une approche double. Tout d'abord, un simulateur paramétrable sera mis en œuvre pour modéliser le système, permettant ainsi de générer des résultats visuels. Cette simulation servira à évaluer des métriques essentielles telles que le temps moyen de traitement et le nombre moyen de clients. En parallèle, une étude analytique sera entreprise en se basant sur le théorème de Jackson, visant à déterminer les conditions de stabilité du système. Les résultats anticipés incluent une comparaison entre les résultats obtenus par simulation et ceux issus de l'analyse théorique, en mettant un accent particulier sur la convergence du système. La durée de simulation a été définie de manière à garantir l'atteinte d'un régime permanent avant la clôture de la période définie, assurant ainsi la pertinence des résultats obtenus.

Simulation :

Dans cette section, un programme JAVA paramétrable est conçu pour simuler le système de base de données distribuée. Le programme intègre un générateur de requêtes selon une loi de Poisson et suit le schéma du réseau de files d'attente du projet. Il permet d'ajuster des paramètres tels que le nombre de serveurs, les temps de traitement, le taux d'arrivée des requêtes, et les probabilités de redirection et de sortie. Les résultats des simulations, incluant le temps moyen de traitement et le nombre moyen de clients, sont visualisés à l'aide de fichiers GnuPlot pour une évaluation graphique du système et des résultats de fonctions réalisées pour ce fait. Cette partie vise à offrir des perspectives concrètes sur le comportement du système, complémentaires à l'étude analytique.

Classes :

File d'attente :

La classe "**FileAttente**" a été implémentée en Java pour représenter une file d'attente. Cette classe utilise une structure de données de type `LinkedList` pour stocker les requêtes en attente de traitement. Le constructeur initialise une file d'attente vide lors de la création de l'objet.

Les principales méthodes de la classe incluent :

- **estVide()** : Cette méthode vérifie si la file d'attente est vide en vérifiant la taille de la `LinkedList`.
- **ajouterRequete(Requete r)** : Cette méthode permet d'ajouter une requête à la file d'attente.
- **getRequeteIndex(int i)** : Cette méthode renvoie la requête à l'index spécifié dans la file d'attente. Si la file est vide, elle retourne `null`.
- **retirerRequete()** : Cette méthode retire et renvoie la première requête de la file d'attente en utilisant la méthode `poll()` de `LinkedList`. Si la file est vide, elle retourne `null`.

- **getFile()** : Cette méthode renvoie la LinkedList représentant la file d'attente.
- **getSize()** : Cette méthode renvoie la taille actuelle de la file d'attente.

MED :

La classe "MED" représente un système de traitement dans le contexte de la simulation d'une base de données distribuée. Cette classe sert de classe mère pour d'autres sous-systèmes tels que les serveurs ou le coordinateur, partageant des fonctionnalités communes. Les principales caractéristiques incluent le taux de service (μ), la période de simulation (D), des listes pour enregistrer les instants d'arrivées, de départs, et d'autres statistiques relatives au système.

- La classe comporte des méthodes pour ajouter et retirer des requêtes de la file d'attente, enregistrer des données à chaque instant, et sauvegarder ces données dans des fichiers pour une analyse ultérieure. Elle offre également des méthodes pour générer des fichiers de configuration GnuPlot facilitant la visualisation des données.
1. Constructeur **MED(double μ , double D)** : Le constructeur initialise un objet MED avec les taux de service μ et la période de simulation D . Il initialise également des listes pour suivre les instants d'arrivées, de départs, le nombre de requêtes en file d'attente, le nombre de requêtes traitées, etc.
 - 2. Méthode **ajouterRequete(Requete r)** : Cette méthode permet d'ajouter une requête à la file d'attente du système.
 - 3. Méthode **retirerRequete()** : Cette méthode retire une requête de la file d'attente du système.
 - 4. Méthode **record()** : Cette méthode enregistre les données à chaque instant, telles que le nombre de requêtes en file d'attente, le nombre total de requêtes traitées, et le nombre de requêtes sorties. Ces données sont stockées dans des listes pour une utilisation ultérieure.
 - 5. Méthode **nbRequetesToFile(String filename)** : Sauvegarde le nombre de requêtes en file d'attente à chaque instant dans un fichier spécifié.
 - 6. Méthode **clientsArriveesToFile(String filename)** : Sauvegarde les instants d'arrivée des clients dans un fichier spécifié.
 - 7. Méthode **clientsDepartsToFile(String filename)** : Sauvegarde les instants de départ des clients dans un fichier spécifié.

- 8. Méthode **saveToFile(String filename, List<Integer> data)** : Méthode générique pour sauvegarder une liste d'entiers dans un fichier spécifié.
- 9. Méthode **saveToFileD(String filename, List<Double> data)** : Méthode générique pour sauvegarder une liste de nombres à virgule flottante dans un fichier spécifié.
- 10. Méthode **saveToFile(String filename, HashMap<Double, Double> data)** : Méthode spécifique pour sauvegarder les données d'une HashMap dans un fichier spécifié.
- 11. Méthode **genererFichierPLT(...)** : Génère un fichier de configuration GnuPlot pour visualiser les données telles que le nombre de requêtes en file d'attente, le nombre total de requêtes traitées, etc.
- 12. Méthode **genererGraphe(String nomFichier)** : Exécute GnuPlot avec le fichier de configuration spécifié pour générer le graphe correspondant.

Les fonctionnalités notables incluent la gestion des requêtes dans la file d'attente, le suivi du nombre de requêtes traitées, en attente, et sorties. De plus, elle fournit des méthodes pour sauvegarder ces données dans des fichiers et générer des graphiques GnuPlot pour une analyse visuelle du comportement du système.

Coordinateur :

La classe **Coordinateur** étend la classe **MED** et représente le coordinateur d'un système de base de données distribuée. Voici une description des principales caractéristiques et méthodes de cette classe :

Attributs :

- **q** : Liste des probabilités que la requête soit redirigée vers chaque serveur.
- **fsi** : Tableau de serveurs dans le système.
- **bdd** : Instance de la classe **BDD** représentant la base de données distribuée.
- **en_cours_de_traitement** : Indique si le coordinateur est actuellement en train de traiter une requête.
- **tempsDebutTrait** et **tempsTraitement** : Enregistrent le temps de début et la durée du traitement en cours.

- **id** : Identifiant du coordinateur.

Constructeurs:

- **Cordinateur(BDD bdd, double mu, double D)** : Initialise le coordinateur avec la base de données distribuée, le taux de service μ , et la période de simulation D .
- **Cordinateur(BDD bdd, double mu, List<Double> q, double D)** : Initialise le coordinateur avec la base de données distribuée, le taux de service μ , les probabilités de redirection q , et la période de simulation D .

Méthodes:

- **setFsi(Server[] fsi)** : Définit le tableau de serveurs.
- **getId()** : Renvoie l'identifiant du coordinateur.
- **getStatut()** : Renvoie le statut du coordinateur (en cours de traitement ou non).
- **getNbClientsMoyen()** : Calcule le nombre moyen de clients dans la file d'attente du coordinateur sur une période spécifique.
- **direction()** : Choix aléatoire d'un serveur vers lequel rediriger la requête en fonction des probabilités définies.
- **redirection()** : gère la redirection des requêtes vers les serveurs. Elle sélectionne de manière aléatoire un serveur en fonction des probabilités définies, retire une requête de la file d'attente du coordinateur, l'ajoute au serveur sélectionné, met à jour la position de la requête, et incrémente le nombre de requêtes traitées. Cette méthode est appelée lorsque le temps de traitement d'une requête est terminé.
- **traitement()** : Méthode principale qui simule le traitement des requêtes dans un système de base de données distribuée. Elle vérifie si des requêtes sont en attente, gère le traitement en cours (bloque le coordinateur le temps de traitement), initialise un nouveau traitement si nécessaire, et redirige les requêtes vers les serveurs. La méthode prend en compte le temps de traitement, la file d'attente, et s'assure de rediriger les requêtes de manière appropriée.

Cette classe modélise le comportement du coordinateur dans un système distribué, prenant en compte les probabilités de redirection et coordonnant le traitement des requêtes entre les serveurs.

Serveur i :

La classe Server représente un serveur dans le système. Chaque serveur est caractérisé par un identifiant unique, un taux de traitement (μ), une probabilité de rediriger une requête vers le coordinateur (p), une probabilité de rediriger une requête vers la sortie (q), et une file d'attente de requêtes.

Les principales méthodes de la classe Server incluent :

- `direction()` : Cette méthode détermine aléatoirement la direction dans laquelle une requête doit être redirigée, soit vers le coordinateur (avec une probabilité p), soit vers la sortie (avec une probabilité q).
- `redirection()` : La méthode joue un rôle essentiel dans la gestion du flux de requêtes au sein du serveur. Lorsqu'elle est appelée, elle utilise la méthode `direction()` pour déterminer aléatoirement si une requête doit être redirigée vers le coordinateur (1) ou vers la sortie du système (0). En cas de redirection vers le coordinateur, la requête est retirée de la file d'attente du serveur, ajoutée à celle du coordinateur, et la position de la requête est mise à jour avec l'identifiant du coordinateur. Ensuite, la méthode `traitement()` du coordinateur est appelée. En revanche, si la direction est vers la sortie du système, le temps de sortie de la requête est enregistré, la position est mise à jour avec "OUT", et la requête est ajoutée aux enregistrements de sortie du système. Les compteurs tant au niveau du système que du serveur pour les requêtes traitées et sorties sont également actualisés. Enfin, l'état de traitement du serveur est réinitialisé à faux, indiquant

qu'aucune requête n'est en cours de traitement. Ces actions coordonnées assurent une simulation réaliste du traitement et de la redirection des requêtes au sein du serveur en fonction de critères aléatoires et probabilistes.

- **traitement()** : Cette méthode gère le traitement des requêtes dans le serveur.

La méthode **traitement()** effectue une vérification initiale pour vérifier si la file d'attente n'est pas vide et que le temps de simulation n'est pas encore fini, ensuite une vérification pour déterminer si le serveur est actuellement en train de traiter une requête, en examinant le drapeau **en_cours_de_traitement**. Si le serveur n'est pas en cours de traitement, la méthode procède à la génération d'un temps de traitement aléatoire pour la requête, utilisant la formule $1.0 / \mu$, où μ représente le taux de traitement du serveur. Ensuite, elle enregistre l'heure de début du traitement dans la variable **tempsDebutTrait** et met le serveur dans l'état de traitement en modifiant la valeur de **en_cours_de_traitement** à **true**. La méthode poursuit ensuite en vérifiant si le temps de traitement est écoulé. Si c'est le cas, elle appelle la méthode **redirection()**, contribuant ainsi à la gestion du flux de requêtes en fonction du temps de traitement généré de manière aléatoire.

Requete :

La classe **Requete** représente une requête dans le système. Elle possède les attributs suivants :

Attributs :

- **id**: Identifiant de la requête.
- **dateEntree**: Instant d'entrée de la requête dans le système.
- **dateSortie**: Instant de sortie de la requête du système.
- **position**: Position actuelle de la requête dans le système.

Méthodes :

- **getId()**: Renvoie l'identifiant de la requête.
- **getDateEntree()**: Renvoie l'instant d'entrée de la requête dans le système.
- **getDateSortie()**: Renvoie l'instant de sortie de la requête du système.

- **getTempsTotal():** Calcule et renvoie la durée totale de traitement de la requête en soustrayant l'instant d'entrée de l'instant de sortie.
- **getPosition():** Renvoie la position actuelle de la requête dans le système.
- **setId(String id):** Modifie l'identifiant de la requête.
- **setDateEntree(double dateEntree):** Modifie l'instant d'entrée de la requête dans le système.
- **setDateSortie(double dateSortie):** Modifie l'instant de sortie de la requête du système.
- **setPosition(String position):** Modifie la position actuelle de la requête dans le système.

La classe *Requete* fournit ainsi des méthodes d'accès et de modification pour interagir avec les attributs d'une requête dans le système.

Generateur de requetes :

La classe **GenerRequetes** est responsable de la génération aléatoire de requêtes dans le système, en respectant un taux d'arrivée défini par le paramètre *lambda*. Elle maintient un compteur de requêtes générées, une liste de requêtes en attente, et une liste de requêtes déjà traitées et sorties du système. Les principales méthodes comprennent:

- **ajouter():** Ajoute une nouvelle requête à la file d'attente du coordinateur après avoir généré aléatoirement le temps entre les arrivées et enregistre le moment prévu de l'arrivée.
- **generer():** La méthode est responsable de la génération aléatoire de requêtes dans le système, en respectant le taux d'arrivée défini par le paramètre *lambda*. Elle commence par vérifier si le système peut accepter de nouvelles requêtes en comparant le nombre actuel de requêtes dans le système avec la capacité maximale définie par le paramètre *D*. Si des requêtes peuvent être ajoutées, la méthode vérifie si le système est actuellement en attente d'une nouvelle requête (drapeau *en_attente*). Si c'est le cas, elle examine si la date d'arrivée prévue de la prochaine requête est atteinte. Si tel est le cas, elle appelle la méthode *ajouter()* pour ajouter la nouvelle requête à la file d'attente du coordinateur.

Si le système n'est pas en attente d'une nouvelle requête, la méthode génère aléatoirement le temps entre les arrivées (`tempsEntreArrivees`) selon une distribution exponentielle inversée. Elle enregistre ensuite le moment de début de traitement (`tempsDebutTrait`) et passe le système en mode d'attente. Si le temps entre les arrivées est inférieur à 1 et supérieur ou égal à 0, elle appelle également la méthode `ajouter()` pour ajouter la nouvelle requête à la file d'attente du coordinateur.

- `tempsMoyen()`: Calcule le temps moyen total que les requêtes ont passé dans le système, y compris le temps de traitement et le temps d'attente.
- `tempsMoyenRegimeStationnaire()`: Calcule le temps moyen que les requêtes ont passé dans le système après avoir atteint le régime stationnaire, en ignorant les premières périodes de démarrage.
- `addReqSortie(Requete r)`: Ajoute une requête traitée à la liste des requêtes déjà sorties du système.
- `saveToFile2(String filename, String filename2)`: Sauvegarde les données de toutes les requêtes (temps total) dans un fichier, prêt pour la génération de graphiques.
- `saveToFile(String filename, String filename2)`: Sauvegarde les données des requêtes arrivées dans le régime stationnaire et sorties du système dans un fichier, prêt pour la génération de graphiques.
- `genererFichierPLT(String filename, String titreImage, int id)`: Génère un fichier de script GNUPlot pour créer un graphique représentant le temps que chaque requête a passé dans le système (en millisecondes) avec différentes catégories.

BDD:

La classe principale BDD est au cœur de la simulation du système distribué. Elle représente la base de données du système, coordonnant le comportement du coordinateur (Fc), des serveurs (fsi), et du générateur de requêtes (generateurR). Voici une explication plus détaillée des différentes parties de cette classe

Attributs :

- **lambda**: Taux d'arrivées global dans le système.
- **muc**: Taux de traitement dans le coordinateur.
- **mui**: Liste des taux de traitement individuels des serveurs.
- **pi**: Probabilités de redirection des requêtes sortantes des serveurs vers le coordinateur.
- **qi**: Probabilités de redirection des requêtes sortantes du coordinateur vers les serveurs.
- **n**: Nombre de serveurs dans le système.
- **fsi**: Tableau représentant les serveurs.
- **Fc**: Coordinateur du système.
- **generateurR**: Générateur de requêtes.
- **D**: Période de simulation.
- **count, nbRequetesSorties, nbRequetesSystem, nbRequetesTot**: Compteurs pour suivre différents aspects de la simulation.
- **tempsMoyenReq, nbReq, nbReSort, nbReEntr**: Listes pour enregistrer différentes mesures au fil du temps.
- **id**: Identifiant de la BDD.

Méthodes :

Constructeurs :

BDD(int id, int n, int D, double lambda, double mu, List<Double> mui, List<Double> qi, List<Double> pi): Constructeur principal pour initialiser les paramètres du système.

BDD(int id, int n, int D, double lambda, double mu, double p, int nbServRapides): Constructeur alternatif avec des paramètres simplifiés (pi, qi).

Méthodes de configuration :

setLambda(double lambda), setMuC(double mu), setMui(List<Double> mui), setPi(List<Double> pi), setQi(List<Double> qi): Méthodes pour définir les paramètres du système.

setElmentMui(int index, double m), setElmentPi(int index, Double p), setElmentQi(int index, double q): Méthodes pour modifier des éléments spécifiques dans les listes.

Méthodes d'accès (Getters) :

Nombreuses méthodes pour récupérer les valeurs des attributs et les résultats de la simulation (getId(), getCount(), getNbRequetesSorties(), etc.).

Méthodes de traitement :

record(): Enregistre différentes mesures telles que le temps moyen de traitement, le nombre de requêtes, etc... à chaque instant i de la simulation .

incrementCount(), incrementNbRequetesSystem(), decrementNbRequetesSystem(), incrementNbRequetesSorties(), incrementNbRequetesTot(): Méthodes pour mettre à jour les compteurs.

generateGraphs(): Génère différents graphiques pour visualiser les résultats de la simulation.

getL(): Calcule le nombre moyen de clients en régime stationnaire dans l'ensemble du système.

getW(): Calcule le temps moyen qu'un client passe dans le system en régime stationnaire.

Méthode principale de simulation :

traitement(boolean generate) : La méthode traitement de la classe BDD constitue le cœur de la simulation du système distribué. Cette méthode coordonne le déroulement de la simulation en itérant sur une période définie (D). À chaque itération, la méthode enregistre différentes mesures, telles que le temps moyen de traitement des requêtes,

le nombre de requêtes dans le système, le nombre de requêtes sorties, et le nombre total de requêtes. Elle génère également de nouvelles requêtes à l'aide du générateur dédié (`generateurR`). La méthode orchestre ensuite le traitement des requêtes par le coordinateur (`Fc`) et chacun des serveurs (`fsi`). Les enregistrements des performances de chaque entité du système sont effectués, et les compteurs sont mis à jour en conséquence. Ce processus se répète jusqu'à ce que la période de simulation soit terminée. En fin de simulation, si spécifié par le paramètre `generate` , la méthode déclenche la génération de graphiques représentant graphiquement les résultats obtenus au cours de la simulation, offrant ainsi une visualisation claire des performances du système distribué. En résumé, la méthode `traitement` encapsule la logique temporelle de la simulation, coordonnant l'évolution du système et enregistrant les métriques clés pour une analyse ultérieure.

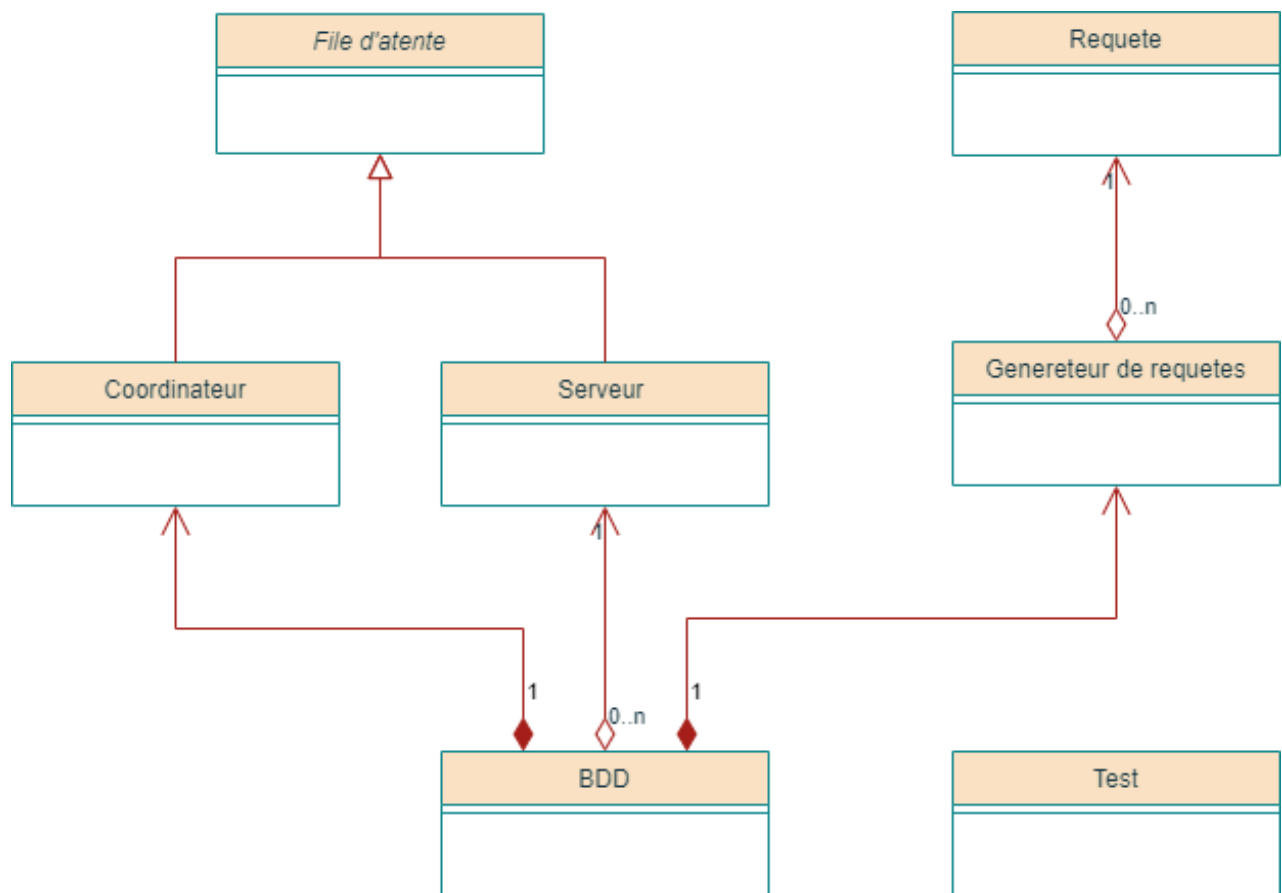
Test :

La classe `Test` en Java est un programme qui effectue des simulations et des analyses de performances sur un système distribué représenté par la classe `BDD` . Voici une explication succincte des principales méthodes et de la logique du programme :

- **tempsMoyenLambda(int nbBDD, int n, int T, double muC):** Cette méthode simule plusieurs instances du système BDD en faisant varier le taux d'arrivée des requêtes (λ) et mesure le temps moyen de passage des requêtes en régime stationnaire. Les résultats sont enregistrés dans des fichiers et utilisés pour générer des graphiques.
- **tempsMoyenN(int nMax, int T, double lambda, double muC):** Similaire à la méthode précédente, mais cette fois, elle fait varier le nombre de serveurs (n) dans le système.
- **tempsMoyen_N_Lambda(int nbBDD, int nMax, int T, double lambda, double muC):** Cette méthode effectue une double variation, faisant varier à la fois le taux d'arrivée des requêtes (λ) et le nombre de serveurs (n). Les résultats sont enregistrés dans des fichiers et utilisés pour générer des graphiques.
- **test(BDD b):** Une méthode utilitaire qui affiche des informations détaillées sur les performances d'une instance spécifique de la classe BDD . Elle affiche le nombre total de requêtes, le nombre de requêtes restant dans le système, le nombre de requêtes sorties, l'efficacité du système, et des détails sur le coordinateur et les serveurs.
- **clear():** Une méthode pour effacer la console, ce qui améliore la lisibilité des résultats lors de l'exécution du programme.
- **system1():** Cette méthode permet à l'utilisateur de spécifier les paramètres du système (nombre de serveurs, période de simulation, taux d'arrivée des requêtes, etc.) à l'aide d'une interface graphique (JFrame).
- **system1Default():** Une version par défaut de la méthode system1() avec des paramètres prédéfinis.

- **system2():** Similaire à `system1()` , mais conçu pour une configuration différente de la classe `BDD` .
- **system2Default():** Version par défaut de la méthode `system2()` .
- **main(String[] args):** La méthode principale du programme. Elle crée une instance de la classe `Test` et exécute différentes simulations, générant ainsi des résultats et des graphiques pour l'analyse des performances du système distribué. Vous pouvez également trouver des appels à d'autres méthodes de simulation dans cette méthode.

L'ensemble de ces classes forment le schéma suivant :



Interface graphique :

Une interface graphique est prévue pour simplifier les tests et la configuration du système, offrant une flexibilité pour définir le nombre de serveurs et leurs paramètres (q_i , p_i , μ_i). Elle permet également d'ajuster le taux d'arrivée λ et d'autres variables essentielles pour la simulation .

The screenshot shows a graphical user interface for configuring a simulation. The window has a title bar with a small icon and the text 'Configuration', and a close button (X) in the top right corner.

The main area is divided into two sections. The top section contains four input fields for global simulation parameters, each with a label to its left:

- nb Servers (N)**: 3
- Temps de Simulation (T)**: 100000
- lambda**: 0.0095
- μ Cordinateur**: 0.1

To the right of these fields is a button labeled 'Créer'.

The bottom section is a table for configuring individual servers. It has three columns: a server identifier, a parameter label, and a value. The first three rows are visible:

Server	Parameter	Value
S0	q	0.3333333333333333
S1	q	0.3333333333333333
S2	q	0.3333333333333333

Each row also includes a small icon (a square with a diagonal line) and a unit symbol (μ) next to the value. To the right of the table, there are three more input fields, each with a label 'p' and a value '0.3333'.

At the bottom of the window is a button labeled 'Valider'.

Tests et Résultats

Scénarios :

Scénario 1 : $\lambda = 1/100$, $p = 1/4$, $n = 2$, nb Serveurs lents : 1.

Scénario 2 : $\lambda = 0.0095$, $p = 1/3$, $n = 3$, nb Serveurs lents : 1.

Soit le temps de simulation $T = 100\ 000$.

Scénario 1 :

- * nb requetes total entrées : 1006
- * nb requetes encore dans le system : 278
- * nb requetes sorties du system : 728
- * * Efficacité : 72,37%
- * * E/S : 1,38%

Cordinateur :

Fc (Coordinateur)

- * nb rqts en file d'attente : 0
- * nb rqts en traitées : 1269
- * rqts en cours de traitement 0

Serveurs :

Fs0

- * nb rqts en file d'attente : 182
- * nb rqts traitées : 495
- * rqts en cours de traitement 1

Fs1

- * nb rqts en file d'attente : 96

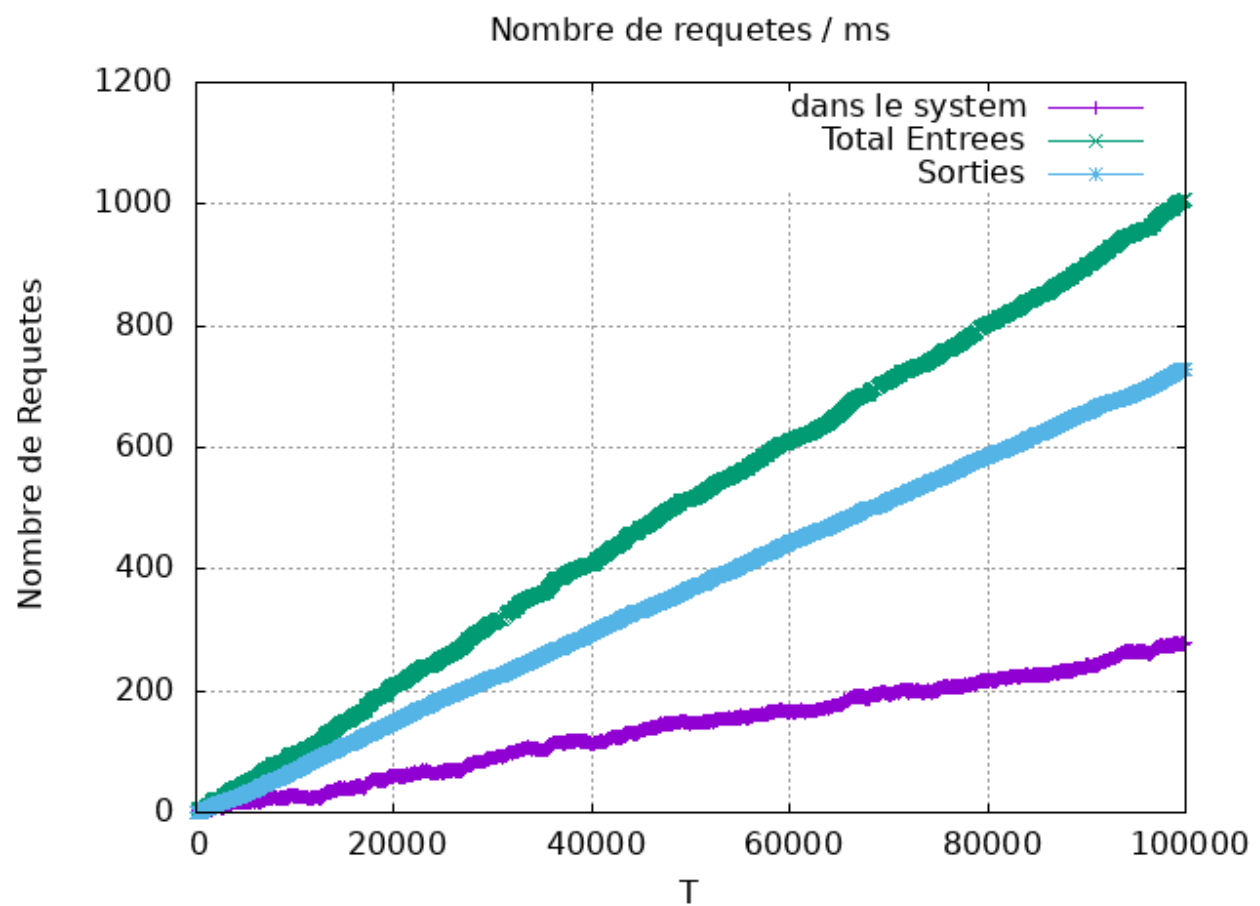
* nb rqts traitées : 496

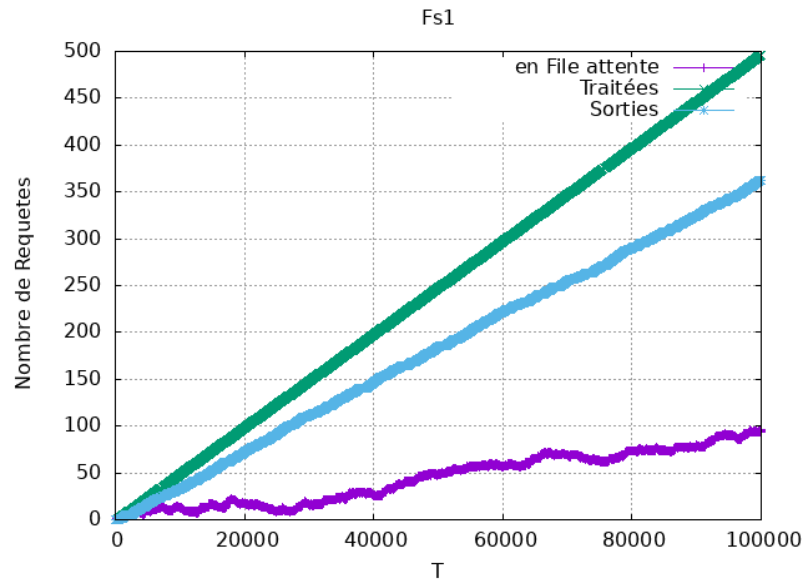
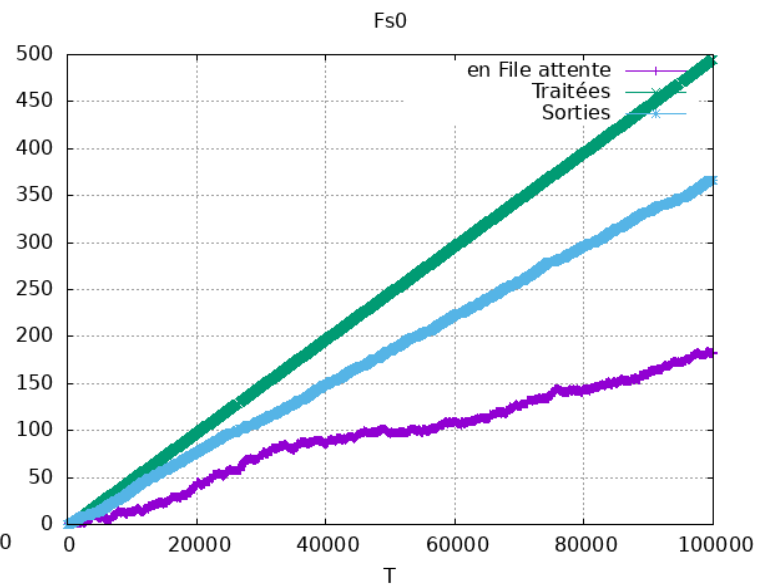
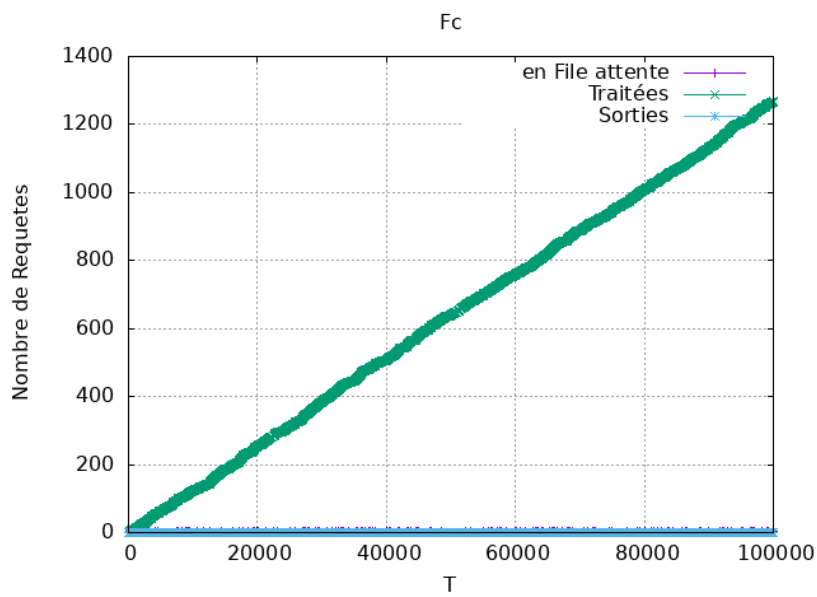
* nb rqts en cours de traitement 1

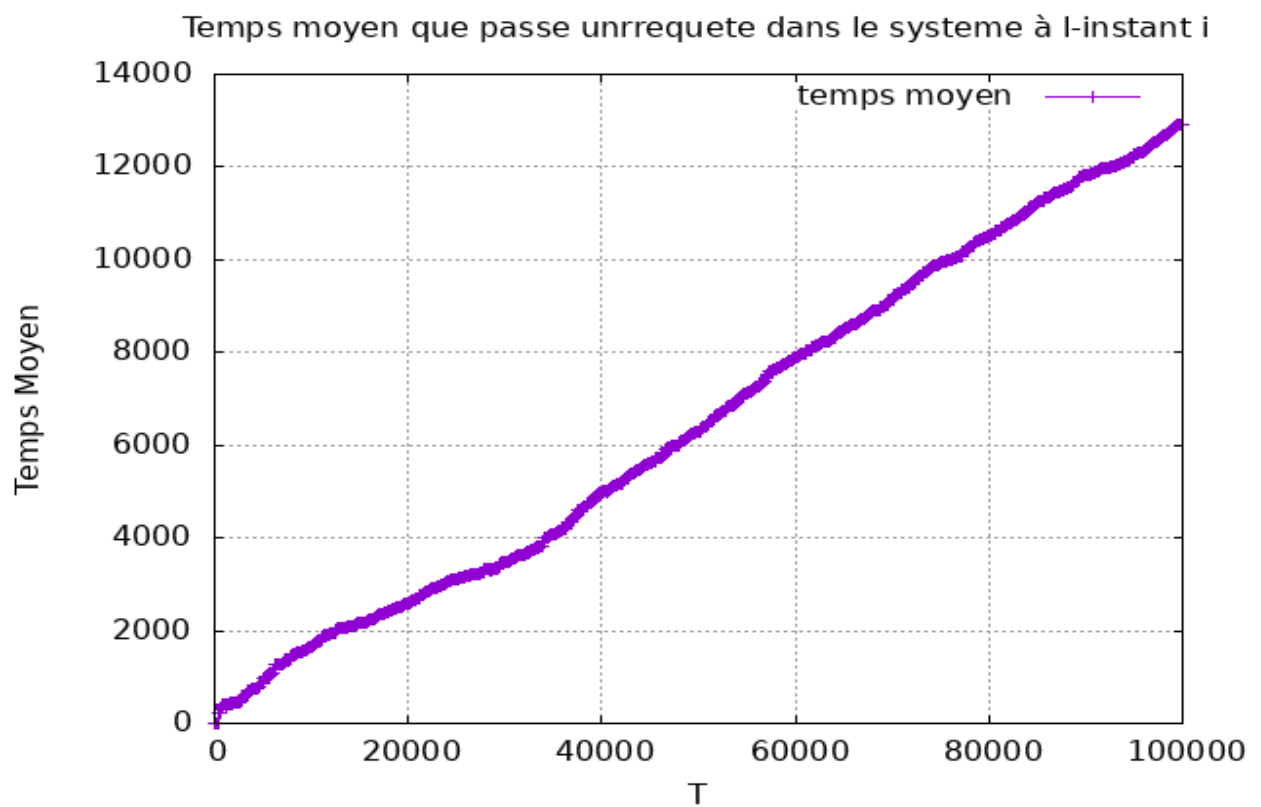
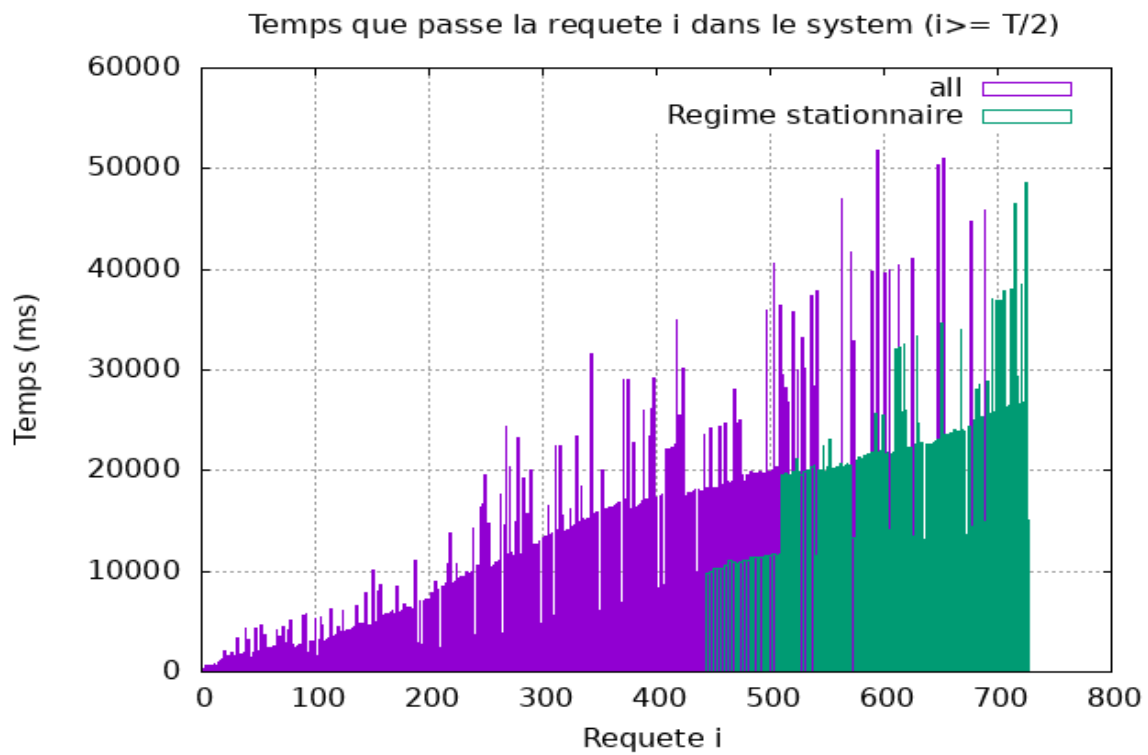
Le temps moyen que passe une requete dans le systeme(RS) est : 18791.220609636846 ms

nb moyen de clients L (RS) = 204.0

Résultats graphiques :







Observations :

Pour ce scénario on observe une croissance linéaire dans le nombre de requêtes dans le system jusqu'à atteindre 278 requêtes à la fin de la simulation .

comme on observe une croissance linéaire dans nombre de requêtes dans la file d'attente des deux serveurs qui atteint 182 dans le serveur lent et 96 pour le rapide .

Une croissance linéaire dans le temps que passe une requête dans le système .

Explication :

File d'attente des serveurs :

Serveur lent : La file d'attente du serveur lent atteignant 182 et reste en croissance linéaire indique que ce serveur est confronté à un trafic important et n'est pas en mesure de traiter les requêtes aussi rapidement qu'elles arrivent. Cela peut entraîner des retards significatifs dans le traitement des requêtes.

Serveur rapide : Bien que le serveur rapide ait une file d'attente plus faible (96) et reste en croissance linéaire également , cela suggère qu'il n'est pas suffisamment rapide pour traiter les requêtes entrantes.

Contribuant ainsi à une croissance linéaire du nombre de requêtes dans le système qui indique que le système n'est pas capable de traiter les requêtes à un rythme suffisamment rapide pour les éliminer rapidement. Cela est dû à un déséquilibre entre le taux d'arrivée des requêtes et la capacité de traitement du système.

L'augmentation du temps que passe une requête dans le système est causée par l'augmentation du nombre de requêtes dans le système. Cette croissance du nombre de requêtes conduit à une surcharge des files d'attente des serveurs, rallongeant ainsi les délais d'attente et contribuant à une augmentation du temps total que chaque requête passe dans le système.

Scénario 2 :

- * nb requetes total entrées : 962
- * nb requetes encore dans le system : 28
- * nb requetes sorties du system : 934
- ** Efficacité : 97,09%
- ** E/S : 1,03%

Coordinateur :

Fc (Coordinateur)

- * nb rqts en file d'attente : 0
- * nb rqts traitées : 1406
- * rqts en cours de traitement 0

Servers :

Fs0

- * nb rqts en file d'attente : 10
- * nb rqts traitées : 457
- * rqts en cours de traitement 1

Fs1

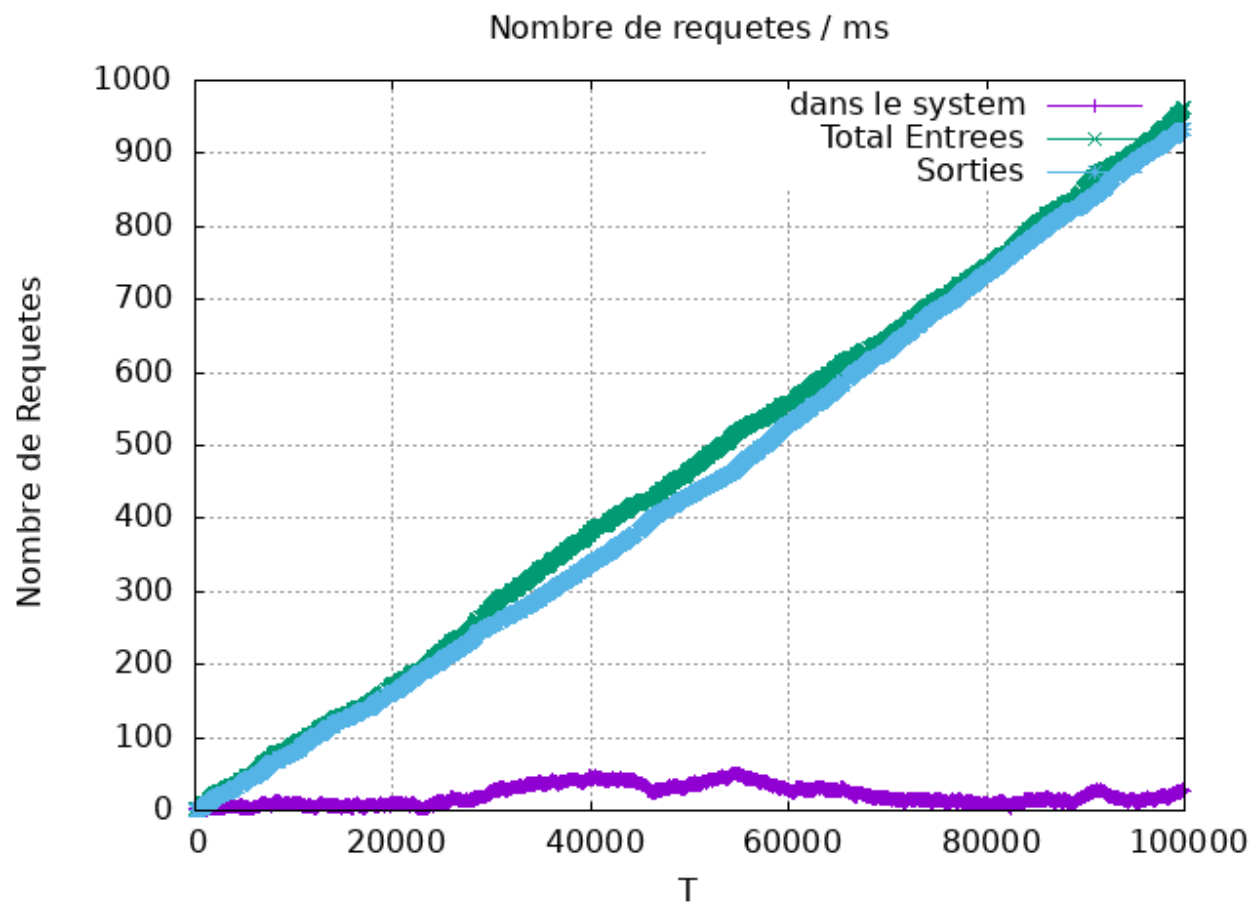
- * nb rqts en file d'attente : 12
- * nb rqts traitées : 481
- * rqts en cours de traitement 1

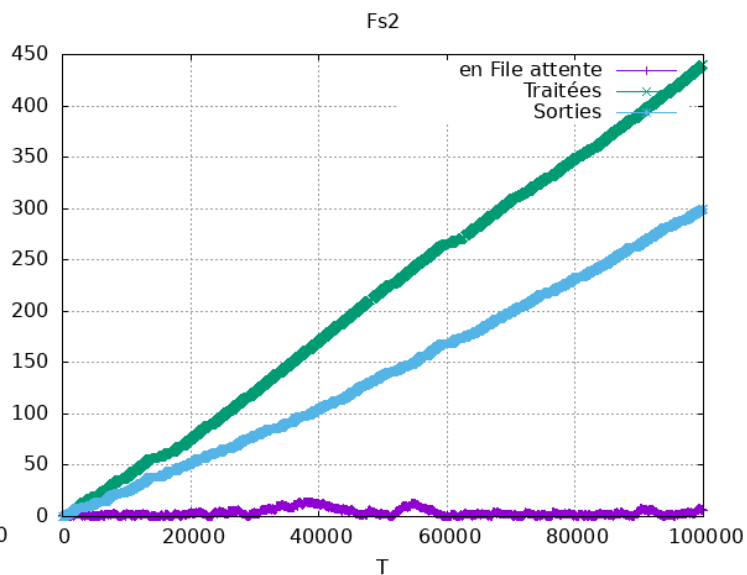
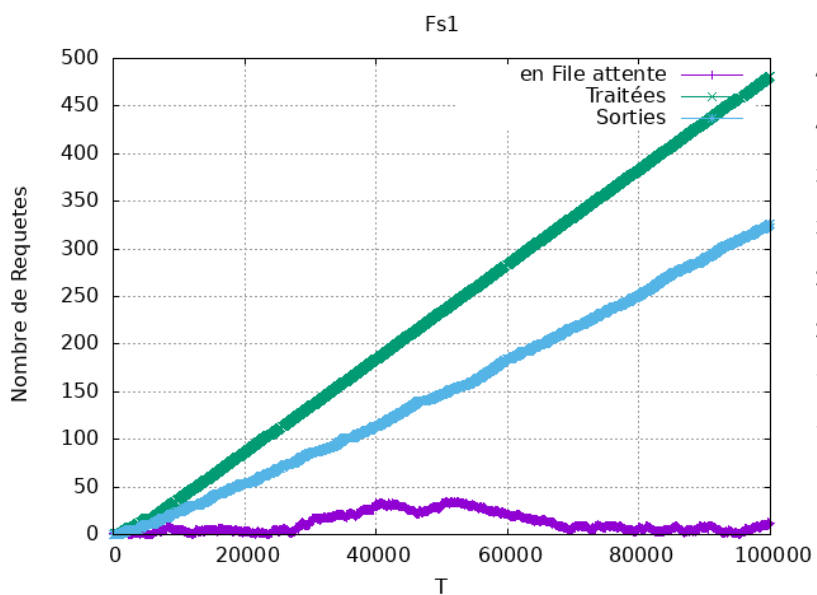
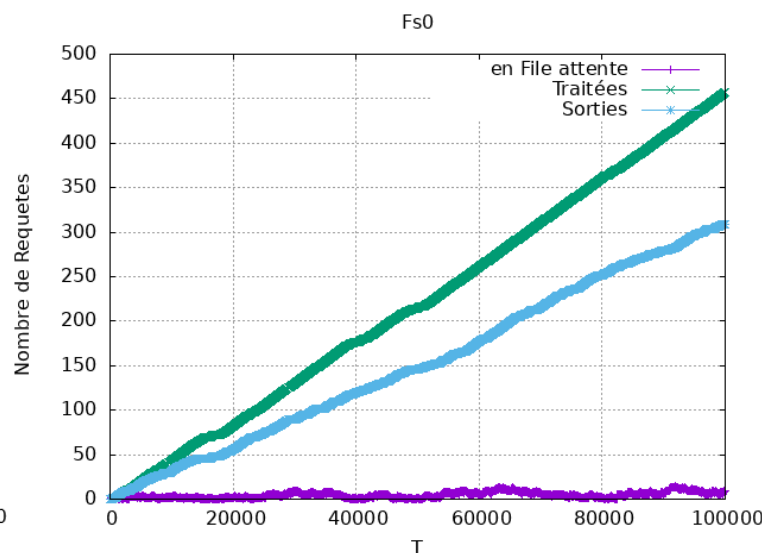
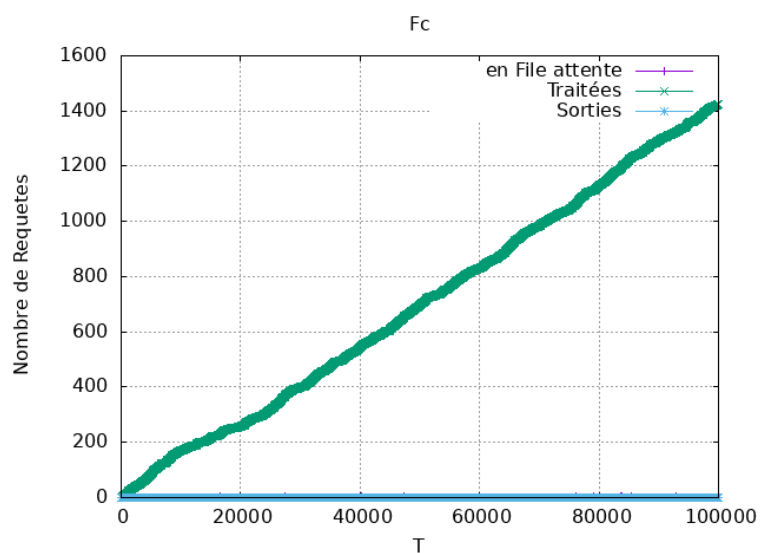
Fs2

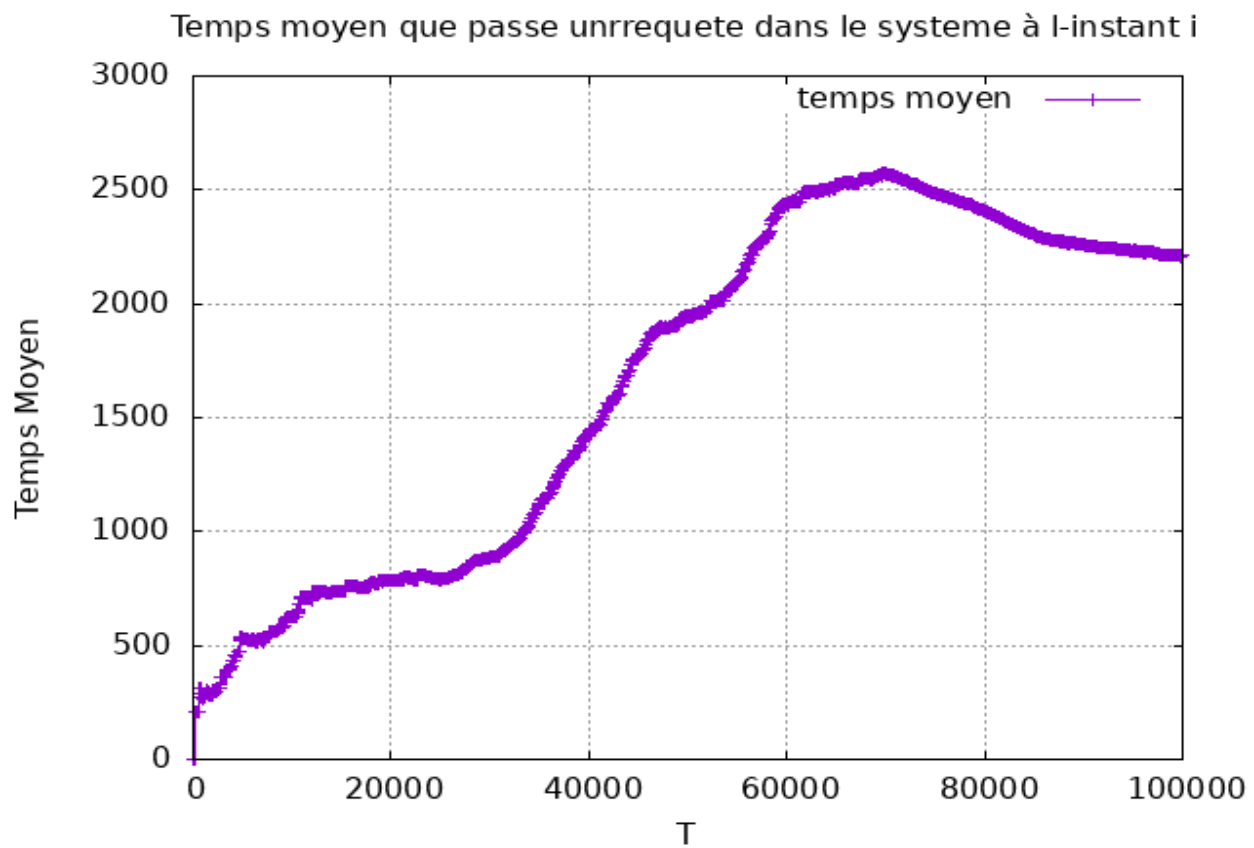
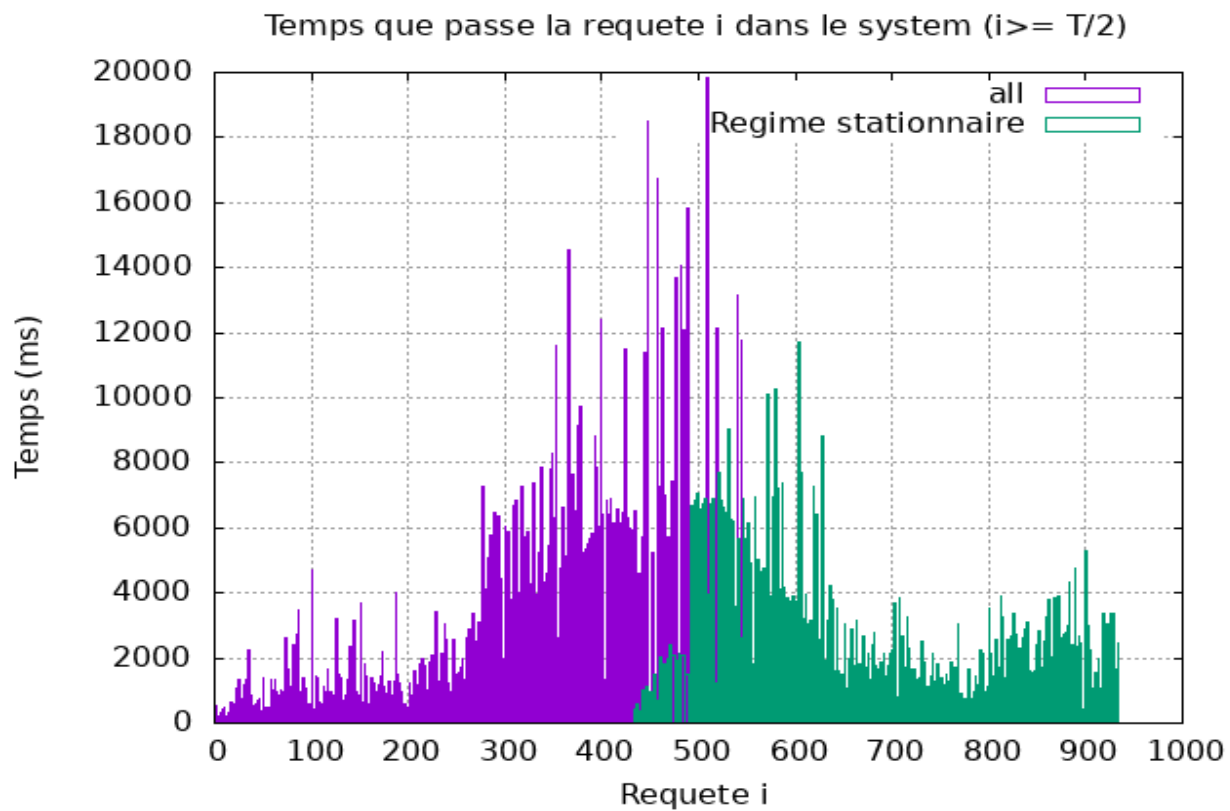
- * nb rqts en file d'attente : 6
- * nb rqts traitées : 440
- * rqts en cours de traitement 1

Le temps moyen que passe une requete dans le système(RS) est : 1951.4064941561319 ms

nb moyen de clients L (RS) = 21.0







Observations :

- La courbe représentant le nombre de requêtes sorties du système est identique à celle du nombre d'entrées : nb rqts sorties du système \approx nb reqs entrées
- Le nombre de requêtes dans le système est quasiment nul .
- les files d'attente des 3 serveurs est presque vides dans le régime stationnaire et tout au long de la simulation.
- Le temps moyen que passe une requête dans le système se stabilise à partir de $T/2$

Explication :

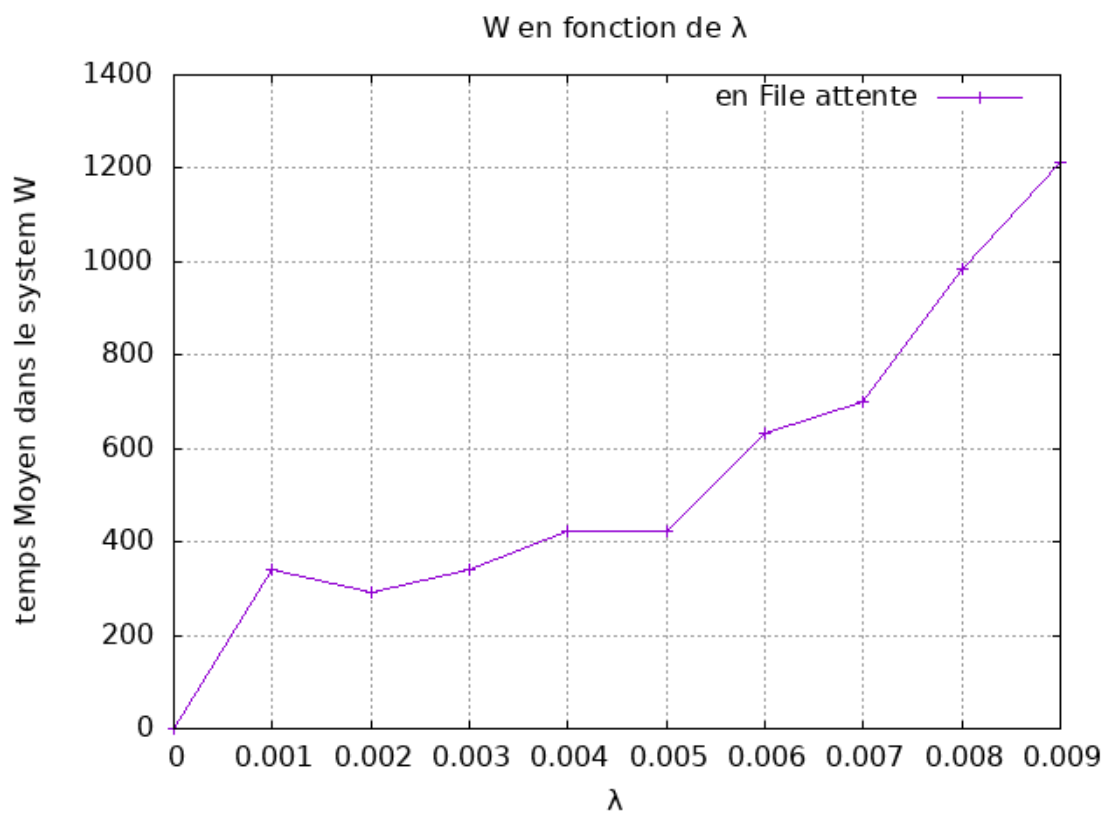
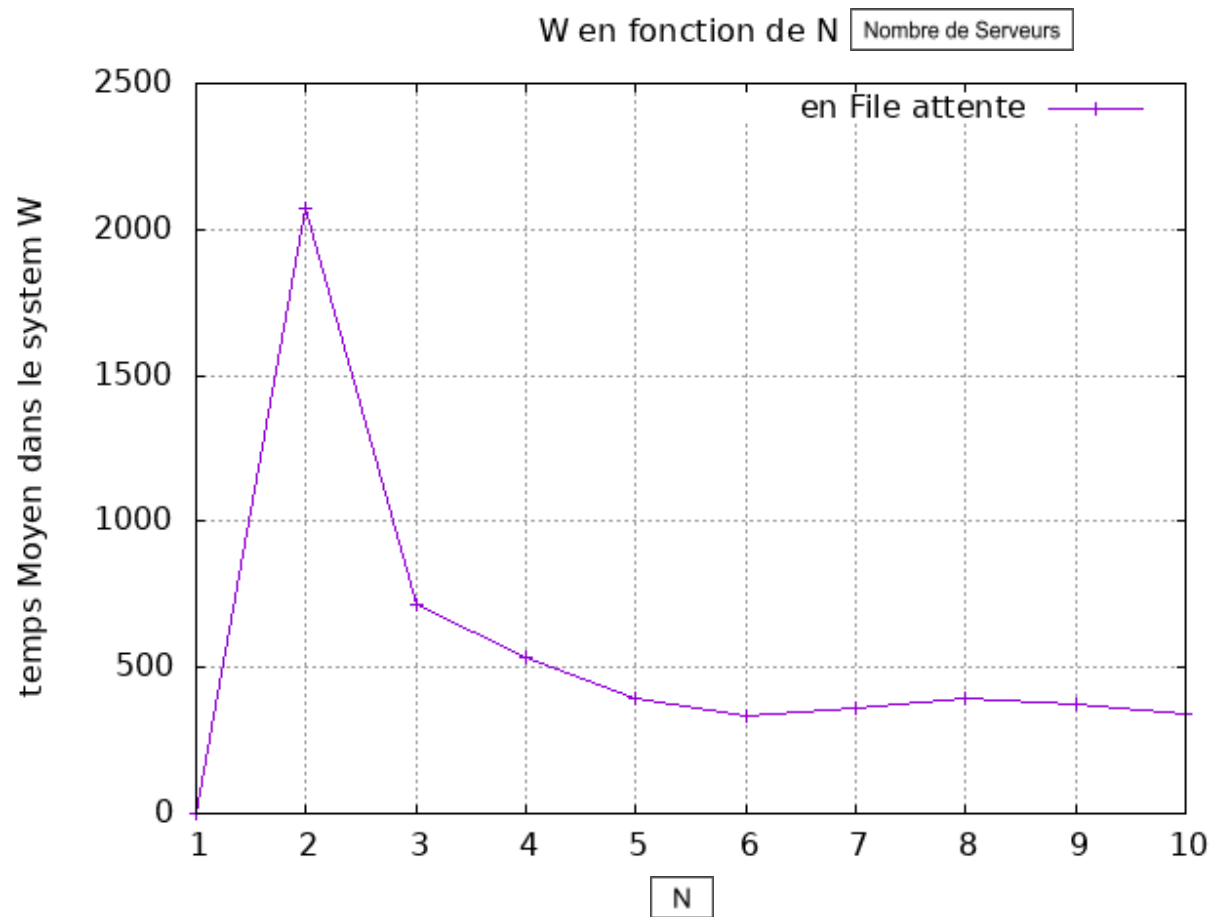
- Équilibre entre les entrées et les sorties : La similarité entre les courbes représentant le nombre de requêtes sorties du système et le nombre d'entrées suggère un équilibre global dans le traitement des requêtes. Cela indique que le système est capable de traiter efficacement un nombre significatif de requêtes entrantes.
- Faible nombre de requêtes dans le système : Le fait que le nombre de requêtes dans le système reste quasiment nul indique que le système fonctionne de manière optimale, traitant les requêtes de manière rapide et efficace, évitant ainsi une accumulation importante de requêtes en attente.
- Files d'attente presque vides : L'observation selon laquelle les files d'attente des trois serveurs sont presque vides dans le régime stationnaire et tout au long de la simulation suggère que les serveurs ont la capacité de traiter les requêtes de manière efficace sans créer de congestion importante dans les files d'attente.
- Stabilisation du temps moyen de traitement : La stabilisation du temps moyen que passe une requête dans le système à partir de $T/2$ suggère que le système atteint un état stable et que les performances du système se stabilisent. Cela peut indiquer que le système a atteint une efficacité optimale dans le traitement des requêtes.

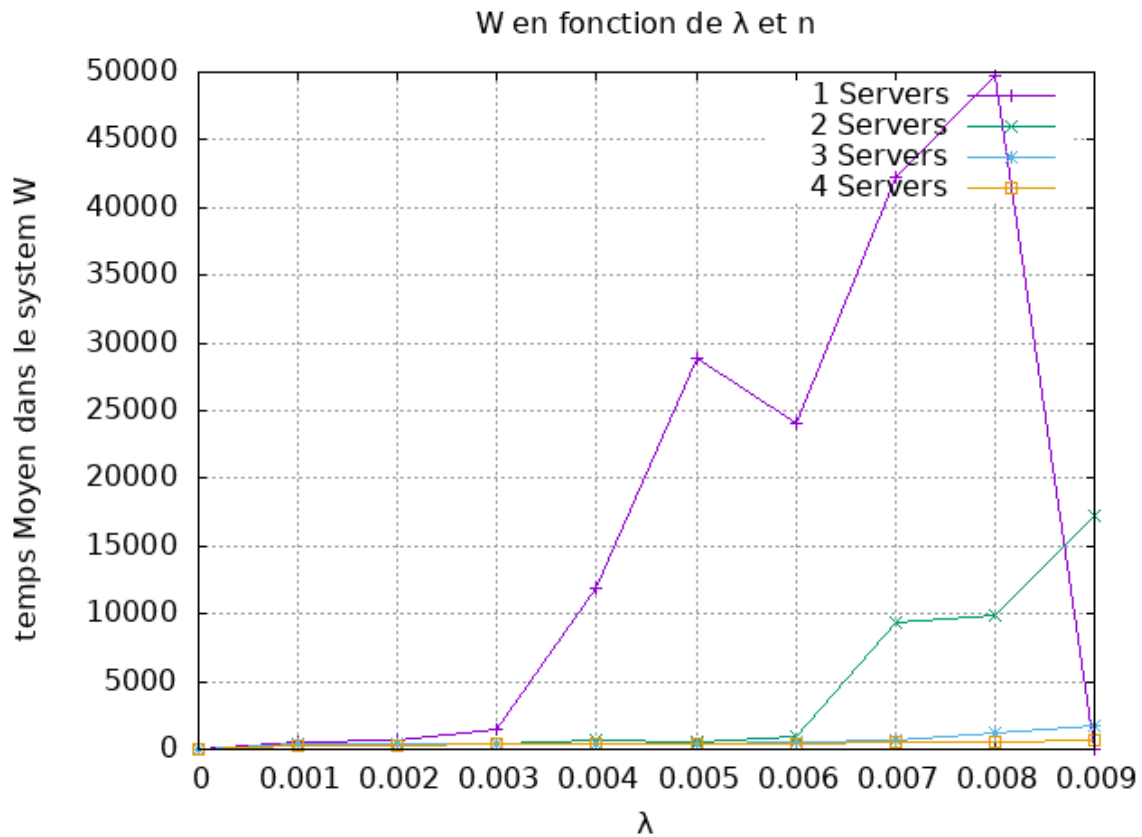
Le deuxième scénario se distingue par sa stabilité, caractérisée par un équilibre entre le nombre de requêtes entrantes et sortantes, des files d'attente minimales et un temps moyen de traitement stabilisé. En revanche, le premier scénario démontre une croissance linéaire du nombre de requêtes dans le système, des files d'attente en expansion, et une augmentation continue du temps moyen de traitement. Cette disparité soulève la question cruciale : quels sont les paramètres spécifiques qui influent sur la stabilité d'un système de traitement de requêtes, et comment peuvent-ils être ajustés pour garantir une performance optimale et une gestion efficace des flux de requêtes? 🤔

Pour explorer cette question, des simulations ont été réalisées pour évaluer l'influence de deux paramètres :

- Le nombre de serveurs
- le taux d'arrivées λ

Résultats :





Observations :

- Réduction du temps de traitement par l'accroissement du nombre de serveurs, et devient stable au bout d'un certain nombre.
- Augmentation du nombre de serveurs par l'augmentation du taux d'arrivée λ .
- L'impact de l'augmentation du paramètre λ est plus significatif sur les systèmes à un petit nombre de serveurs que sur ceux avec un grand nombre de serveurs.
- Certains systèmes avec un grand nombre de serveurs montrent peu d'influence face à une augmentation de λ .

Explications :

Les résultats s'expliquent par la dynamique des systèmes étudiés. En augmentant le nombre de serveurs, on réduit la charge de travail sur chaque serveur, ce qui conduit

généralement à une diminution du temps de traitement global. Cependant, cette amélioration n'est pas linéaire et peut atteindre un plateau au-delà d'un certain nombre de serveurs, où l'efficacité supplémentaire devient marginale.

L'augmentation du taux d'arrivée λ influe sur la charge de travail du système. Pour les systèmes avec un petit nombre de serveurs, une augmentation de λ peut rapidement surcharger les serveurs, entraînant une augmentation significative du temps de traitement. En revanche, les systèmes avec un grand nombre de serveurs ont une capacité accrue pour gérer des taux d'arrivée plus élevés, ce qui les rend moins sensibles à l'augmentation de λ .

Alors, les simulations mettent en lumière l'impact significatif du nombre de serveurs et du taux d'arrivée λ sur les performances des systèmes. Une augmentation du nombre de serveurs contribue à réduire le temps de traitement, mais avec des gains décroissants au-delà d'un certain seuil. Parallèlement, l'augmentation du taux d'arrivée λ influence davantage les systèmes avec un petit nombre de serveurs, tandis que les systèmes plus grands sont moins sensibles à cette variation. Trouver un équilibre optimal entre ces deux paramètres est crucial pour maximiser l'efficacité du système.

Divers autres paramètres influent sur le comportement du système, notamment les probabilités de sortie du système (q_i) qui jouent un rôle essentiel dans la distribution des requêtes entre le coordinateur et les serveurs. Modifier ces probabilités peut avoir un impact significatif sur les performances globales du système, en favorisant par exemple une redirection préférentielle vers les serveurs rapides. De même, en ajustant les probabilités p_i , une augmentation de ces dernières entraîne systématiquement une augmentation du nombre de requêtes sortant du système, contribuant ainsi à une plus grande stabilité du système.

Etude Analytique

En utilisant le théorème de Jackson pour les réseaux ouverts :

les paramètres λ_c et λ_i :

$$\begin{cases} \lambda_i = \lambda_c \cdot q_i & (1) \\ \lambda_c = \lambda + \sum_i (\lambda_c \cdot q_i \cdot p_i) & (2) \end{cases}$$

$$\begin{cases} \lambda_i = \lambda_c \cdot q_i & (3) \\ \lambda_c = \lambda + \lambda_c \cdot \sum_i (p_i \cdot q_i) & (4) \end{cases}$$

$$\begin{cases} \lambda_i = \lambda_c \cdot q_i & (5) \\ \lambda_c = \lambda + \lambda_c \cdot \sum_i (p_i \cdot q_i) & (6) \end{cases}$$

$$\begin{cases} \lambda_i = \lambda_c \cdot q_i & (7) \\ \lambda_c \cdot (1 - \sum_i (p_i \cdot q_i)) = \lambda & (8) \end{cases}$$

$$\begin{cases} \lambda_i = \lambda_c \cdot q_i & (9) \\ \lambda_c = \frac{\lambda}{1 - \sum_i (p_i \cdot q_i)} & (10) \end{cases}$$

$$\begin{cases} \lambda_i = q_i \cdot \frac{\lambda}{1 - \sum_i (p_i \cdot q_i)} & (11) \\ \lambda_c = \frac{\lambda}{1 - \sum_i (p_i \cdot q_i)} & (12) \end{cases}$$

Nombre moyen de requêtes dans le système :

$$\begin{cases} L = L_c + \sum_i L_i \\ L_i = \frac{\lambda_i}{\mu_i - \lambda_i} \end{cases}$$

Temps moyen que passe une requête dns le système :

$$L_i = \frac{L}{\lambda}$$

Condition de stabilité :

$\lambda_i < \mu_i$ dans tous les serveurs y compris le coordinateur .

Application numérique :

Scénario 1 :

$$\lambda = 0.0095$$

$$N = 3$$

$$u_1 = \frac{1}{100}, \quad u_2 = \frac{1}{100}, \quad u_3 = \frac{1}{200}$$

$$p_i = \frac{1}{3}$$

$$q_i = \frac{1}{3}$$

on Obtient

$$\left\{ \begin{array}{l} \lambda_C = 1/75 \\ \lambda_1 = 2/300 \\ \lambda_2 = 2/300 \\ \lambda_3 = 2/300 \end{array} \right. > \mu_2 \quad L_2 = -4!!!!$$

⚠ On a $\lambda_2 > \mu_2$ alors le système n'est pas stable .

Scénario 2 :

$$\lambda = 0.0095$$

$$N = 3$$


$$u_1 = \frac{1}{100}, \quad u_2 = \frac{1}{100}, \quad u_3 = \frac{1}{200}$$

$$p_i = \frac{1}{3}$$

$$q_i = \frac{1}{3}$$

on Obtient

$$\left\{ \begin{array}{l} \lambda_C = 0.01425 \\ \lambda_1 = 0.00475 \\ \lambda_2 = 0.00475 \\ \lambda_3 = 0.00475 \end{array} \right. \quad \left\{ \begin{array}{l} L = 20.92 \text{ req} \\ W = 2202.2 \text{ s} \end{array} \right.$$

On a pour tout i $\lambda_i < \mu_i$ d'où le système est **Stable** .

Les résultats issus de notre analyse théorique concordent de manière significative avec les observations tirées de la simulation, tant en ce qui concerne le temps moyen passé par une requête dans le système que le nombre moyen de requêtes présentes dans le système.

Conclusion

En résumé, notre étude approfondie du système de gestion de bases de données distribuées a révélé des insights significatifs sur son comportement dynamique. Nous avons identifié des facteurs clés tels que le **nombre de serveur N** , le **taux d'arrivées λ** , le **taux de traitement μ_i** des serveurs, ainsi que les probabilités de sortie du système (**p_i**) et d'acheminement entre les serveurs (**q_i**), qui exercent une influence substantielle sur la stabilité et l'efficacité du système. Les simulations ont mis en évidence des tendances intéressantes, telles que la stabilisation du temps moyen de traitement avec **l'augmentation du nombre de serveurs**. Nous avons également souligné l'importance de l'ajustement des probabilités **q_i** , qui peut jouer un rôle critique dans la répartition des requêtes entre le coordinateur et les serveurs, en **favorisant les serveurs rapides** pour améliorer les performances du système. De même, les probabilités **p_i** de sortie du système ont été identifiées comme des éléments clés contribuant à l'amélioration des performances en augmentant le taux de sortie du système en augmentant leurs valeurs. Tout comme l'augmentation du taux de traitement μ_i des serveurs, accélérant ainsi le traitement. Un autre aspect crucial est le taux d'arrivée λ , où une **réduction** de ce taux accélère le système, en baissant le nombre de requêtes rejoignant le système. Les résultats théoriques ont confirmé la cohérence avec les observations pratiques, renforçant la validité de notre analyse. En conclusion, cette étude offre des perspectives éclairantes pour la conception et l'optimisation des systèmes de gestion de bases de données distribuées, soulignant l'importance d'une prise en compte judicieuse de divers paramètres pour garantir des performances optimales.