Université Nice SOPHIA ANTIPOLIS

L3 Miage, 29 novembre 2019

TP-1 Réseaux : Simulation de la file M/M/1

File M/M/1 :
- arrivées poissonniennes de taux λ
- durée de service exponentielle de taux μ
- 1 seul serveur
- buffer d'attente de taille infinie
- discipline de service FIFO

1 Objectifs pédagogiques

L'objectif de ce TP est de simuler l'évolution du nombre de clients dans une M/M/1 en utilisant le langage de programmation C. Nous étudierons l'influence du facteur de charge $\rho = \lambda/\mu$.

2 Consignes

General : Lisez TOUT l'énoncé avant de taper une seule ligne de code.

Ressources : Vous pouvez utiliser les matériels du cours (slides de CM, TPs passés, ...) ainsi que toute ressource disponible sur Internet.

3 Simulation à Evénements Discrets

La Simulation à Evénements Discrets (SED) nous permet de modéliser des buffers des noeuds des commutation et de simuler leurs fonctionnements. La SED est un type de simulation qui fait progresser l'horloge en étapes discrètes, souvent de taille irrégulière. Ces étapes discrètes correspondent à l'intervalle de temps ente deux événements consécutifs. Donc, la simulation est dirigée par des événements :
- A chaque étape, l'horloge avance vers l'événement suivant planifié dans une file d'attente d'événements, et l'événement est traité.
- Étant donné que ce sont les événements les seuls qui peuvent entraîner un changement de l'état de la simulation, il n'y a aucun intérêt à faire avancer l'horloge dans des pas de temps plus petits que les intervalles entre les événements.

Les simulations à événements discrets sont utilisées de façon classique pour modéliser des problèmes d'accès concurrents à une ressource partagée, c.-à-d. les problèmes des files d'attente. Dans notre cas, lorsque plusieurs paquets de données (les clients de notre système) essaient d'accéder à la bande passante (la ressource partagée) du lien de sortie. Donc, on aura affaire à deux types d'événements :
- **Arrivée** du paquet au système et entrée dans la file d'attente (suivi, si possible, par le début du service).
- **Départ** du paquet du système, après la fin du service (suivi, si possible, par le début du service d'un paquet mis en attente au préalable dans le buffer).

4 Le lien de sortie d'un noeud comme une file M/M/1

- Les paquets de données correspondent aux clients.
- Le serveur (ou service) corresponde à un lien de transmission avec un débit r (bps) par lequel des paquets de données sont transmis
- Le temps de service moyen (t_s) correspond à la durée de transmission (d_{trans}) sur le lien de sortie : t_s = s (bits) / r (bps) = 1 / μ (s).
- Le file d'attente est infinie.

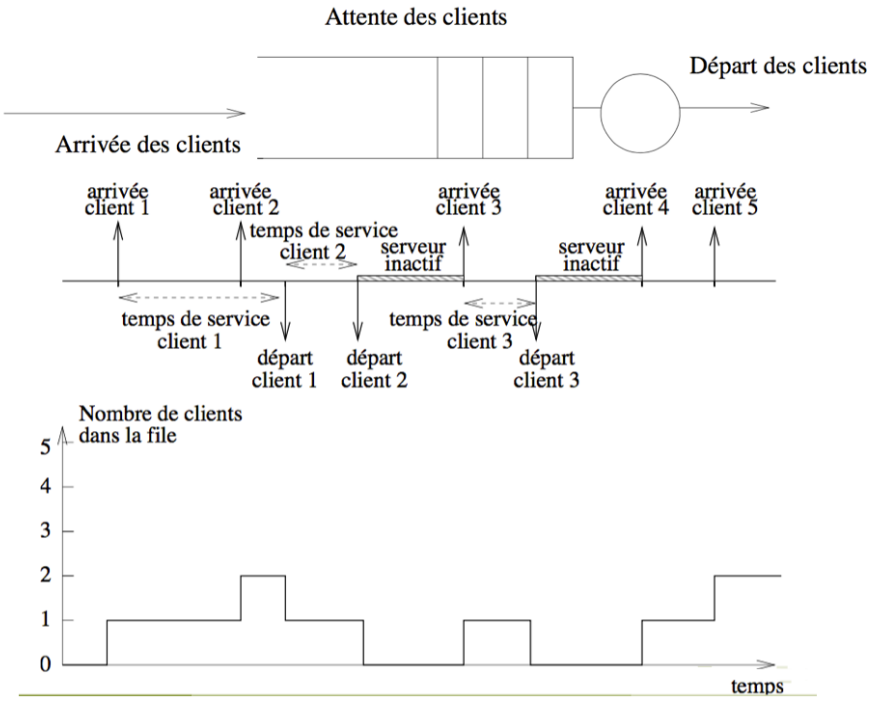

FIGURE 1 – Modèle de file d'attente

— Le processus des arrivées de paquets est un processus de Poisson de paramètre λ. Cela veut dire que la durée des inter-arrivées (temps entre deux arrivées consécutives) suit une loi exponentielle de paramètre λ, où λ représente le nombre moyen d'arrivées par unité de temps (unité : paquets/seconde) et, en consequence, la durée moyenne des inter-arrivées vaut 1/ λ (unité : seconde).
— La durée de service suit une loi exponentielle de paramètre μ, où μ représente le nombre moyen de départs (terminaisons de services) par unité de temps (unité : paquets/seconde) et, en consequence, la durée moyenne de service vaut 1/ μ (unité : seconde).

5 Programme C

Ecrivez un programme `C` simulant l'évolution au cours du temps du nombre N de clients dans le système en suivant les étapes proposées.

5.1 Architecture générale du simulateur

Vous trouverez le code de base dans le repertoire `TPs/TP1/src`. Il s'agit des fichiers (i) `main.c`, qui code la boucle principale du simulateur ; et, (ii) `queue.c`, qui code les fonctionnes auxiliaires, notamment celles qui rajoutent / effacent des elements dans les deux *listes* qui constituent la base du simulateur : (i) la *liste des événements* (arrivées et departs de paquets), ou *Future Even List, FEL* ; et (ii) le *buffer* (tampon) de la file d'attente où on stocke les paquets en attente d'accéder au lien de sortie du noeud.

5.2 Fonctionnes auxiliaires

Dans le fichier entête (fichier `queue.h`), vous trouverez les prototypes des fonctionnes à employer. La plupart sont déjà implementées sauf `popBuffer`. Les trois plus importantes sont responsables de gérer la *liste des événements* et le *buffer* de sortie :

— `popEvent` : Il enleve l'evenement qui est à la tête de la *liste des événements*.

— `popBuffer` : Il enleve le paquet qui est à la tête du *buffer* de sortie, c.-à.-d., met en service ce paquet : lui envoie par le lien de sortie.
— `insertEvent` : Il introduit un nouvel événement soit dans la *liste des événements*, soit dans le *buffer* de sortie, le cas echéant.

Les elements dans la *liste des événements* et dans le *buffer* de sortie sont représentés comme des listes chaînées des structures `node` :

```
typedef struct node {
  int callerID; // ID du paquet
  char event; // Char 'A' ou 'D' pour une arrivée ou un depart respect.
  double time; // Temps auquel l'événement aura lieu
  struct node *next; // Pointer vers le suivant événement
} node_t;
```

Notez qu'une liste est ainsi composé des structures `node` où le champ *next* pointe vers le suivant element. Grace à ces pointeurs, on parcourra les listes. Mais, pour commencer à parcourir la liste, on a besoin d'identifier la tête de la liste. Pour cela, on aura un pointeur `node_t * head` vers l'adresse du premier de l'element qui est en tête. De cette manière la notion de `pointeur vers la tête` `node_t * head` et la notion de liste chaînée devient équivalent : chaque liste chaînée est identifiée par sa tête. Regardez dans la fonction `insertEvent` comme on utilise les pointeurs pour parcourir la liste, dans ce cas pour inserter un nouvel élément. Le procédé est montré dans la Fig. 2.

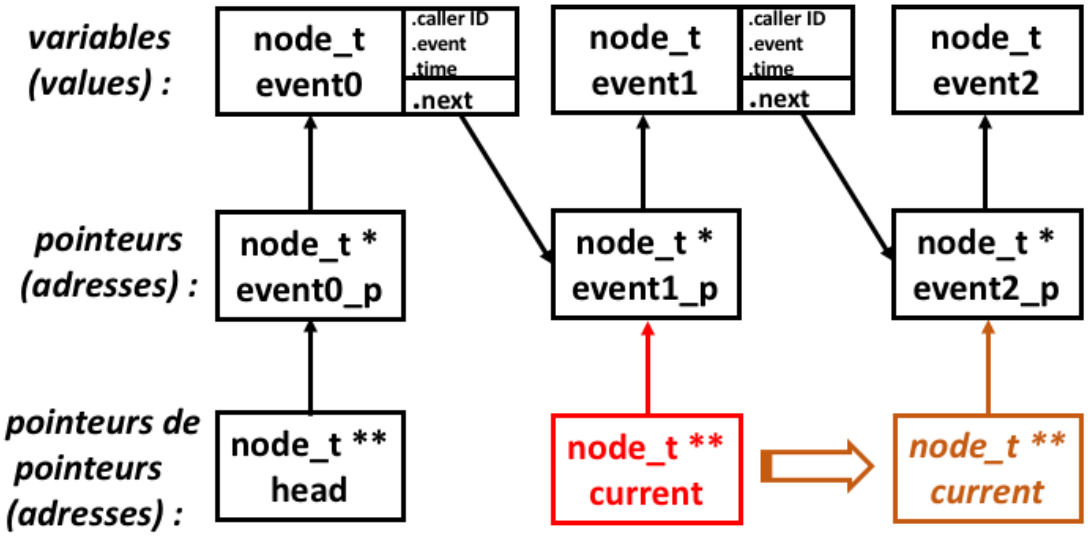

FIGURE 2 – Liste des événements

5.3 La boule principale

Dans la function *main*, la boucle principale est à implementer. Dans chaque iteration, on avance jusqu'au prochain événement (arrivée ou départ). Donc, la boucle doit parcourir la *liste des événements* à partir du pointeur vers la tête `node_t * head` de *liste des événements*. Notez que deux situations se posent en fonction de type d'événement (arrivée ou départ) qui est pointé par `node_t * head`. Si l'événement est une arrivée de paquet, on doit planifier **deux** événements :

— *la prochaine arrivée :* on sait que les interarrivées suivent une loi exponentielle avec un temp moyen de $1/\lambda$ seconds, on peut calculer le moment d'arrivé du suivant paquet et le mettre dans la *liste des événements*.
— *la sort de paquet arrivant* : Deux cas apparaissent en fonction de l'état du système (s'il y déjà des plaquets en attente) :
  — Si le système est vide (il n'y aucun paquet paquet dans le lien de sortie ni implicitement dans le buffer) : le paquet qui arrive peut accéder directement au lien de sortie sans besoin d'attendre dans la buffer. On planifie le depart de ce paquet (loi exponentielle avec un temp moyen de $1/\mu$ seconds) et on le met dans la *liste des événements*.
  — Si le système n'est pas vide (au moins, il y un paquet dans le lien de sortie en train de se transmettre) : on met le paquet qui viens d'arriver dans le buffer. Aucun événement *départ* est planifié puisque le paquet ne peut pas encore accéder au lien de sortie. ATTENTION : le *buffer* de sortie est une autre liste chaînée composé que des structures `node` de type *arrivée* de paquets, c.-à.-d. des paquets arrivées auparavant et mis en queue. Le *buffer* a son propre pointeur vers sa tête : `node_t * bhead`.

Par contre, si l'événement est un départ de paquet, on efface ce départ de la *liste des événements*, on fait avancer la tête `node_t * bhead` du buffer de sortie vers le suivant element et on planifie dans la *liste des événements* le depart du nouveau paquet qui est dans la tête du buffer. Vous devez faire tout cela à l'intérieur de la fonction `popBuffer`.

6 Votre travail

En suivant la description de la gestion de la *liste des événements* et du *buffer* de sortie, qui est décrit dans la section précédente, finissez le simulateur M/M/1. Cela suppose (1) implementer la gestion de l'événement *départ de paquet* à l'intérieur de la fonction `popBuffer` ; et, (2) remplir la boucle principale dans la fonction *main*.

Une fois finie l'implémentation, vérifiez que elle marche correctement. A la fin du code de *main*, des statistiques sont calculées en utilisant (1) des données de la simulation et (2) des formules théoriques vues en cours. Les valeurs prevues par les formules doivent être proches des valuers calculées à partir des données de la simulation. Si c'est pas le cas, modifiez votre code pour faire marcher la simulation.

En supposant que le simulateur marche correctement, procédez aux suivants tests.

Question 1 : Tester l'influence du facteur de charge $\rho = \lambda/\mu$. Prendre une valeur de $\rho < 1$ (par exemple, λ = 1 et μ = 2). Puis prendre une valeur de $\rho > 1$ (par exemple, λ = 3 et μ = 2). Que se passe-t-il dans chacun des deux cas ? ATTENTION : Souvenez vous que le simulateur utilise des temps moyens de interarrivée (1/ λ) et de service (1/ μ) comme des arguments d'entrée.

Question 2 : Répétez l'experiment précédent de manière systématique pour une dizaine de valeurs. Par exemple, gardez fix μ = 2 et faire évaluer λ = {0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0 }. Pour chaque expérience, annotez le délai moyen de séjour dans le système (dans le rapport de sortie "Mean residence time") et tracez ensuite la courbe en fonction du facteur de charge ρ.

Question 3 : A partir de l'allure de la courbe, quel est la composante du délai moyen de séjour d'un paquet dans le système qui domine dans la region de la courbe lorsque $\rho << 1$? Et dans la region de la courbe lorsque $\rho \to 1$? En raison du cours de théorie, vous pouvez expliquer pourquoi ?