Modélisation et Simulation

Simulation à événements discrets

G. Bontempi
Département d'Informatique
Boulevard de Triomphe - CP 212

November 16, 2022

Systèmes à temps discret

Definition

Un système est dit *à temps discret* si l'ensemble T est un ensemble discret. Les systèmes à temps discrets peuvent être

- *synchrones* où les variables du système prennent leurs valeurs selon une fréquence préétablie (par exemple T est l'ensemble des nombres entiers),
- *asynchrones* où les instants de temps suivent une distribution aléatoire. Ces systèmes sont aussi appelés systèmes à évènements discrets.

Simulation à événements discrets

- Simulation d'un système dont l'état ne peut changer que lors d'instants temporels distincts.
- Un événement est la circonstance (entrée) qui permet au système de changer d'état:
  - en absence d'événements l'état resterait indéfiniment le même, ce qui n'était pas forcement le cas dans les systèmes vus jusqu'ici.
  - un événement pourrait ne pas déclencher un changement d'état.
- En théorie, une simulation à événements discrets pourrait être menée de manière manuelle. En pratique l'ordinateur est incontournable.

Les files d'attente

- **File d'attente**: un modèle de l'accès séquentiel d'un ensemble d'**utilisateurs** (par exemple clients) à un nombre limité de **ressources** (par exemple les guichets d'une banque).
- *Client* peut dénoter des personnes, des machines, des voitures, des patients, des paquets des données, des emails, des containers.
- *Service* peut dénoter des employés, des dépanneurs, des mécaniciens, des docteurs, des routers, des systèmes anti-spam, des grues.
- Compromis entre le nombre de services, l'utilisation des services et la satisfaction des clients dans un contexte incertain.

Les files d'attente (II)

- La théorie des files d'attente et/ou la simulation peuvent être utilisées pour prédire la performance du système (par exemple la moyenne du temps d'attente d'un client) en fonction des paramètres (par exemple le nombre des services, la loi des arrivées, la loi des services).
- *File d'attente*: ensemble des clients qui attendent d'être servis, à l'exclusion de celui qui est en train de se faire servir.
- *Système d'attente* l'ensemble des clients qui font la queue, y compris celui qui se fait servir.
- Pour des systèmes de petite taille et pour certaines distributions de probabilité, les mesures de performance peuvent être calculées d'une manière analytique.
- Simulation incontournable quand la complexité et le réalisme du système augmentent.

Propriétés d'une file d'attente

- La population des clients peut être de **taille** finie ou infinie (par exemple les clients d'un restaurant).
  - Dans une population de taille infinie, la loi des arrivées ne dépend pas du nombre de clients qui ont déjà été servis par le système.
- La **capacité** d'un système peut être finie (par exemple le nombre des voitures en attente dans une station d'essence) ou infinie (par exemple le nombre de spectateurs en attente d'un concert).
- Le **temps** entre deux arrivées peut être constant ou aléatoire. Un modèle couramment utilisé pour modéliser une loi aléatoire des arrivées est le processus de Poisson (et donc le temps entre l'arrivée du client $i-1$ et du client i suit la loi exponentielle).

Propriétés d'une file d'attente

- Nous pouvons avoir
  1. différents comportements des clients dans la file d'attente (par exemple partir avant de joindre la file si la file est trop longue, partir après un certain temps d'attente dans la file ou changer la file).
  2. différentes façons de gérer la file: FIFO, LIFO, SIRO (Service In Random Order), SPT (Shortest Processing Time first), PR (service according to PRiority).
- Le nombre des services peut être égal à 1, à un nombre fini ou à un nombre infini (self-service).
- Le temps de service peut être constant ou suivre une distribution de probabilité.

Loi de Poisson

- Supposons qu'un événement (par exemple l'arrivée d'un client) se produise en moyenne $\lambda \in \mathbb{R}$, $\lambda > 0$ fois par unité de temps.
- La loi de Poisson est une loi de probabilité discrète qui décrit le comportement du nombre d'évènements se produisant avec une **fréquence moyenne connue** et **indépendamment du temps écoulé** depuis l'évènement précédent.
- La variable aléatoire discrète $\mathbf{k} \in \mathbb{N}$ représente le nombre de fois où l'événement se produit par unité de temps.
- La v.a. $\mathbf{k}$ suit une loi de Poisson avec paramètre $\lambda > 0$ si

$$\text{Prob}\{\mathbf{k} = k\} = \exp^{-\lambda} \frac{\lambda^k}{k!}, \qquad k = 0, 1, 2, \ldots$$

  pour tout entier naturel k.
- La moyenne et la variance de $\mathbf{k}$ sont égales à λ.

Processus d'arrivée de Poisson

Soit $\mathbf{n}_a(t) = \max_i : t_i \leq t$ le nombre aléatoire d'arrivées à (ou avant) l'instant $t \geq 0$, où t_i est l'instant d'arrivée du $i$ème client.

Definition

Le processus stochastique $\mathbf{n}_a(t)$ est appelé un processus de Poisson si

1. les clients arrivent un à la fois.
2. le nombre d'arrivées dans l'intervalle $(t, t + \Delta t]$, c.-à-d. la v.a. $\mathbf{n}_a(t + \Delta t) - \mathbf{n}_a(t)$ est indépendante de $\mathbf{n}_a(u), 0 \leq u \leq t$.
3. le nombre moyen d'arrivées pendant un intervalle de temps ne dépend que de la durée de l'intervalle
4. la distribution de $\mathbf{n}_a(t + \Delta t) - \mathbf{n}_a(t)$ est indépendante de t pour tout $t > 0, \Delta t > 0$.

Processus d'arrivée de Poisson

Theorem

Si $\mathbf{n}_a(t)$ *est un processus de Poisson alors le nombre d'arrivées*

$$\mathbf{k} = \mathbf{n}_a(t + \Delta t) - \mathbf{n}_a(t)$$

dans chaque intervalle de taille Δt *est une v.a. de Poisson avec paramètre* $\lambda \Delta t$ *avec* $\lambda > 0$, *c.-à-d.*

$$Prob\{\mathbf{k} = k\} = \exp^{-\lambda \Delta t} \frac{(\lambda \Delta t)^k}{k!},$$

où $k = 0, 1, 2, \ldots,$ *et* $t, \Delta t \geq 0$

Poisson et exponentielle

Theorem

Si $\mathbf{n}_a(t)$ est un processus de Poisson avec un taux λ alors les quantités

$$a_i = t_i - t_{i-1}$$

sont i.i.d. et distribuées selon la loi exponentielle $\mathcal{E}(\lambda)$ de paramètre λ.

Si **k** (discrète) suit une loi de Poisson, et traduit le nombre d'apparitions d'un phénomène aléatoire dans un intervalle de temps t (fréquence moyenne λ), alors la variable aléatoire **a** (continue) représente l'intervalle de temps séparant deux apparitions d'un évènement donné (période moyenne $1/\lambda$).

Distribution exponentielle

Une variable continue **a** est distribuée selon une loi de probabilité exponentielle de paramètre $\lambda > 0$ ($\mathbf{a} \sim \mathcal{E}(\lambda)$) si sa densité de probabilité est

$$p_{\mathbf{a}}(a) = \begin{cases} \lambda e^{-\lambda a} & \text{if } a \geq 0 \\ 0 & \text{if } a < 0 \end{cases}$$

- La fonction de répartition est $F_{\mathbf{a}} = \text{Prob}\{\mathbf{a} \leq a\} = 1 - e^{-\lambda a}$
- La moyenne de **a** est $1/\lambda$.
- La variance de **a** est $1/\lambda^2$.
- Dans ce cas $1/\lambda$ est la durée de vie moyenne $\frac{1}{\lambda}$.

Notation des files d'attente

Vu la diversité des possibles configurations des files d'attente, Kendall a proposé la notation

$$A/B/c/N_s/K$$

où

- $A \in \{M, D, E_k, PH, H, G\}$ dénote la distribution du temps entre deux arrivées consécutives. Notons que $A = M$ signifie que la distribution est exponentielle alors que $A = G$ signifie que la distribution est générique.
- B dénote la distribution du temps de service
- c représente le nombre de services
- N_s dénote la capacité du système
- K représente la taille de la population des clients.

Par exemple $M/M/S/\infty/\infty$ dénote une file d'attente FIFO avec une population infinie et S services où les temps de service et les temps entre deux arrivées suivent une distribution exponentielle. Nous ne traiterons que ce cas simple.

Etat

- Si le système a $\mathcal{S}$ services (par exemple $\mathcal{S}$ comptoirs ou guichets) l'état du système pourrait être défini par les variables suivantes:
  1. l'état des services (actif ($B_j = 1$) ou désactivé ($B_j = 0$)): cette variable nous permet de déterminer si le client qui arrive sera servi immédiatement ou devra se mettre en file.
  2. le nombre Q_j de clients en attente du $j$ème service: cette variable nous permettra de définir l'état futur du service quand le service d'un client aura terminé.
  3. le temps a_{ij}, $i = 1, \ldots, Q_j$ d'arrivée de chaque client dans la file.
  4. l'instant du dernier événement.

Notons que si le système était composé par un seul service, l'état d'activité du service pourrait être déduit à partir du nombre de clients dans le système.

Evénements

- L'ensemble des événements est composé par:
  1. l'arrivée d'un client à l'instant t qui est demandeur du service j: cet événement fixe la valeur $a_{ij} = t$ pour le nouveau client et active le $j$ème service $B_j(t^+) = 1$ si $B_j(t^-) = 0$. Si $B_j(t^-) = 1$ alors il se met en attente $(Q_j(t^+) = Q_j(t^-) + 1)$
  2. la fin du service d'un client: ceci cause $Q_j(t^+) = Q_j(t^-) - 1$ et si $Q_j(t^+) = 0$ désactive le $j$ème service $B_j(t^+) = 0$.

Considérations

- Dans l'exemple en question, tous les événements causent un changement d'état. Toutefois ceci n'est pas obligatoire. On pourrait envisager des événements qui causent d'autres effets comme la fin de la simulation, ou le changement de la manière de visualiser les sorties.

Gestion du temps

- La nature dynamique des modèles à événements discrets demande un mécanisme pour simuler l'évolution du temps.
- Il est donc nécessaire de définir une variable (en anglais *simulation clock*) qui stocke le temps virtuel de la simulation.
- Notons qu'aucune relation n'existe entre le temps virtuel de la simulation et le temps d'exécution du programme.
- Deux stratégies peuvent être envisagées pour la gestion du temps de la simulation:
  1. avancement par incrément fixe
  2. avancement jusqu'au prochain événement (*next time event*) et calcul de l'instant du temps auquel l'événement suivant aura lieu.

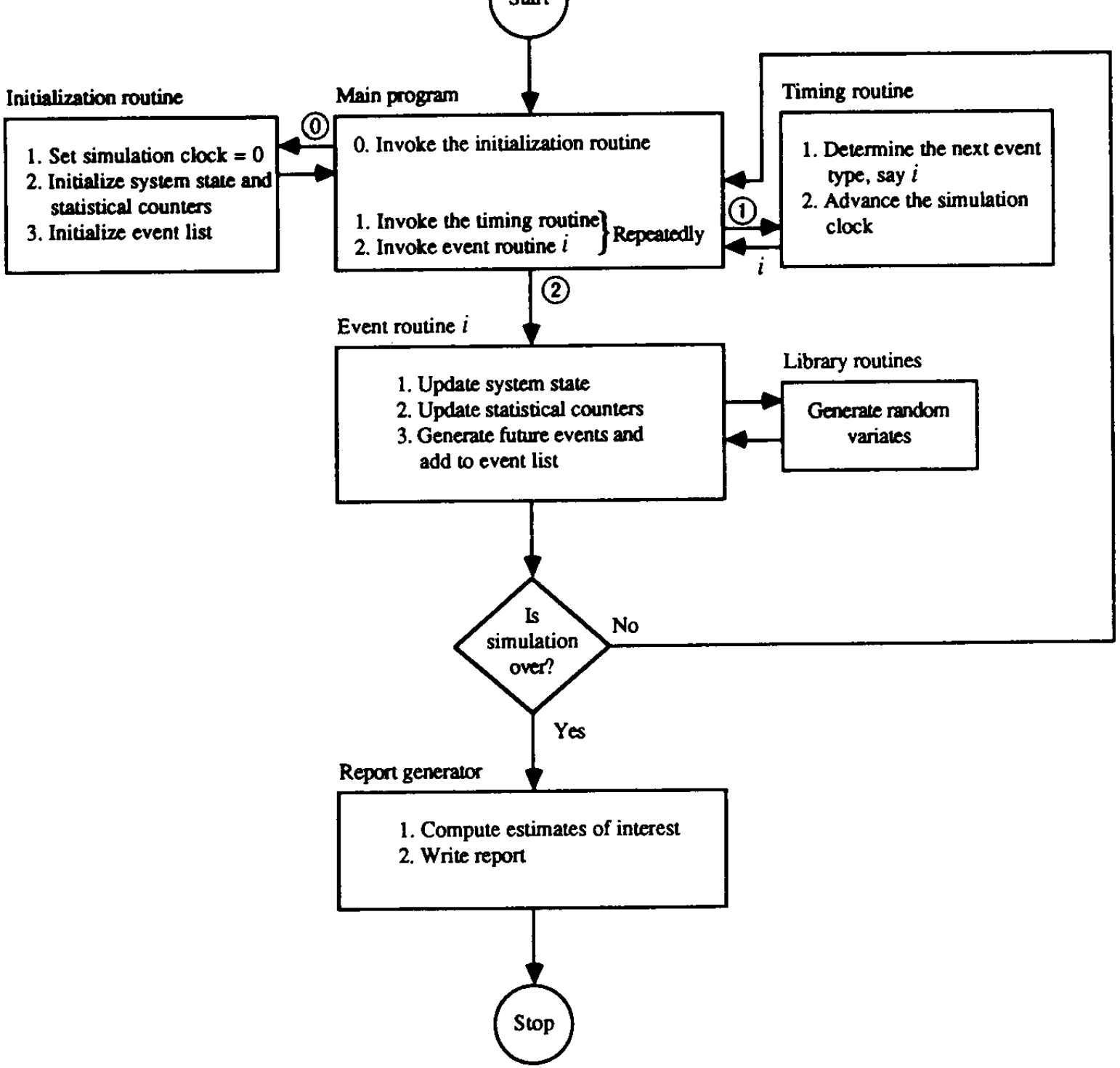
Gestion du temps (II)

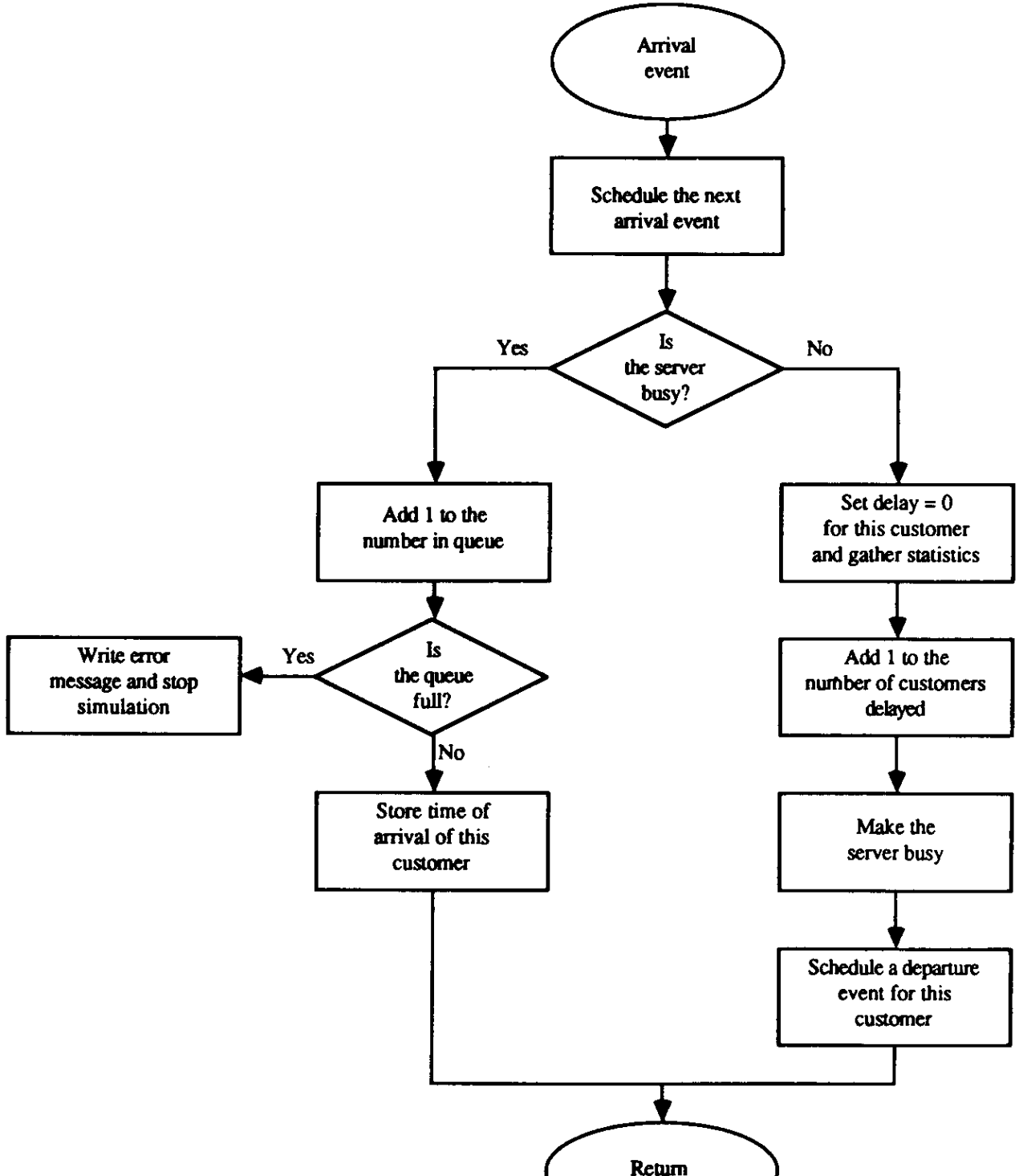
- Une limitation de l'approche par incréments fixes est que si les événements ont lieu seulement à des instants bien précis il est inutile de simuler quand le système est inactif (c.-à-d. quand aucun événement n'a lieu).
- Dans l'approche *next-time* le clock de la simulation est initialisé à zéro au départ. Ensuite la simulation consiste dans la répétition en boucle des opérations suivantes jusqu'à quand une quelconque condition d'arrêt est activée:
  1. le clock est avancé jusqu'à l'instant du prochain événement
  2. l'état du système est mis à jour selon le type d'événement qui a eu lieu.
  3. la liste des événements futurs est mise à jour. La génération des événements futurs demande l'utilisation d'un générateur de nombres aléatoires.

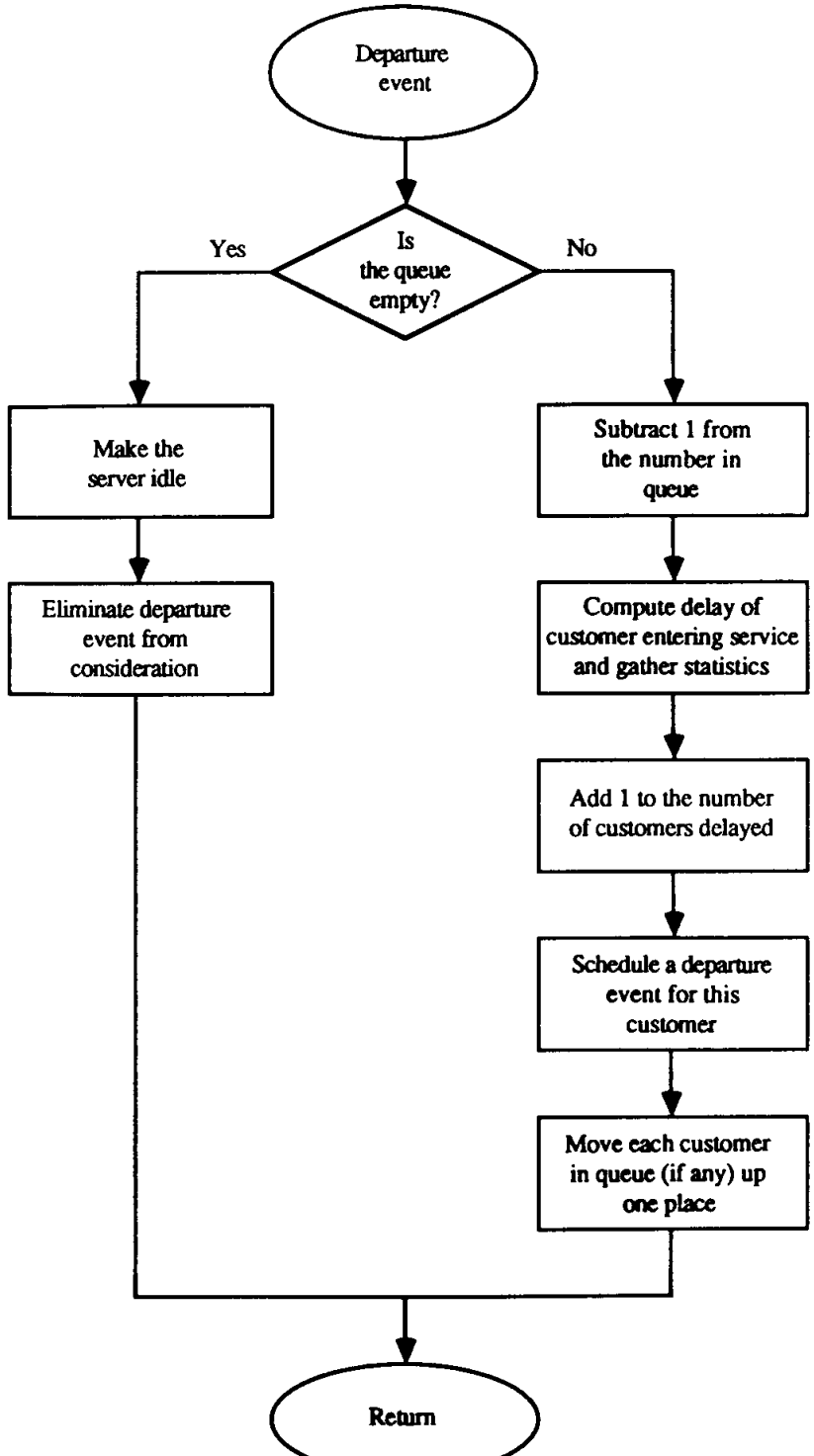
Simulateur à événements discrets

Voici les composantes essentielles d'un simulateur à temps discret:

- Etat du système: ensemble de variables nécessaires et suffisantes pour décrire le système à un certain instant.
- Clock
- Liste d'événements
- Compteurs statistiques: variables utilisées pour mémoriser de l'information sur la performance du système.
- Sous-programme *gestion du temps*: il fait avancer la valeur du clock selon les stratégies mentionnées auparavant.
- Sous-programmes *événement*: ils mettent à jour l'état du système selon le type d'événement qui a lieu.
- Librairies statistiques: elles sont utilisées pour la génération des nombres aléatoires et pour l'analyse statistique des données collectées par les compteurs.
- Générateur de rapport: le sous-programme qui estime à partir des données collectées une série de mesures de performance.


Start
Initialization routine
1. Set simulation clock = 0
2. Initialize system state and statistical counters
3. Initialize event list
Main program
0. Invoke the initialization routine
1. Invoke the timing routine
2. Invoke event routine i
Repeatedly
Timing routine
1. Determine the next event type, say i
2. Advance the simulation clock
Event routine i
1. Update system state
2. Update statistical counters
3. Generate future events and add to event list
Library routines
Generate random variates
Is simulation over?
No
Yes
Report generator
1. Compute estimates of interest
2. Write report
Stop


Arrival event
Schedule the next arrival event
Is the server busy?
Yes
No
Add 1 to the number in queue
Set delay = 0 for this customer and gather statistics
Is the queue full?
Yes
Write error message and stop simulation
No
Add 1 to the number of customers delayed
Store time of arrival of this customer
Make the server busy
Schedule a departure event for this customer
Return


Departure event
Is the queue empty?
Yes
No
Make the server idle
Eliminate departure event from consideration
Subtract 1 from the number in queue
Compute delay of customer entering service and gather statistics
Add 1 to the number of customers delayed
Schedule a departure event for this customer
Move each customer in queue (if any) up one place
Return

Simulation d'une file d'attente

Considérons l'exemple de la file d'attente avec $\mathcal{S} = 1$ décrit auparavant.
Introduisons la notation additionnelle:

- $a_i = t_i - t_{i-1}$: temps entre l'arrivée du $i$ème et du $i-1$ème client.
- S_i: temps du service du $i$ème client
- D_i: temps d'attente dans la file du $i$ème client
- $c_i = t_i + D_i + S_i$: instant de temps auquel le service du $i$ème client est complété et le client s'en va.
- e_i: instant de temps auquel l'$i$ème événement (de nature quelconque) a lieu.
- Q: nombre de clients en attente.
- Y: nombre de clients dans le système. Notons que $Y = Q + 1$ si $Q > 0$.

Soient $F_{\mathbf{a}}$ et $F_{\mathbf{S}}$ les distributions de probabilité des variables aléatoires $\mathbf{a}$ et S, respectivement.

Exécution de la simulation

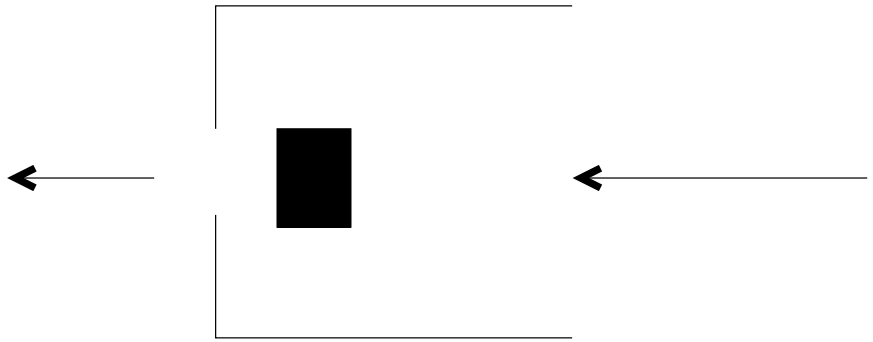
- $t = 0$: le serveur est inactif ($B(0) = 0$) et il n'y a pas de clients en attente ($Q(0) = 0, Y(0) = 0$). L'instant du premier événement e_1 est généré en échantillonnant $\mathbf{a}_1$ de la distribution de probabilité $F_{\mathbf{a}}$ et en posant $e_1 = t_1 = \mathbf{a}_1$.
- Le clock est avancé de $t = 0$ à $t = t_1$.
- $t = t_1$: le server passe d'inactif à actif ($B(t_1^+) = 1$), $Q(t_1^+) = 0$, $Y(t_1^+) = 1$ et $D_1 = 0$. La durée du temps de service du client $i = 1$ est calculée en échantillonnant la distribution de probabilité $F_{\mathbf{S}}$. On obtient:

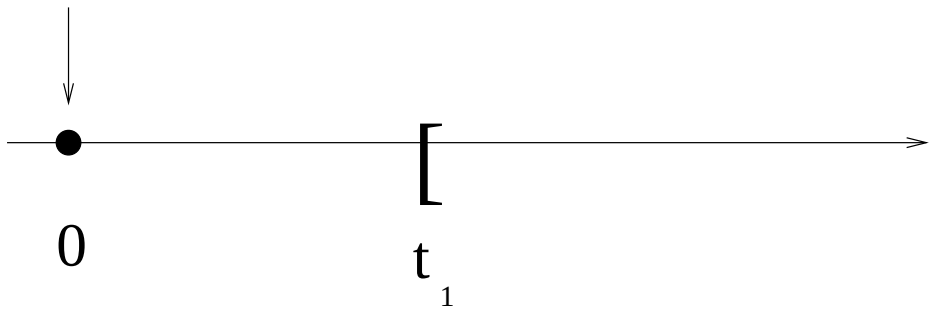
$$c_1 = t_1 + D_1 + S_1 = t_1 + S_1$$

Le temps d'arrivée du deuxième client est généré en échantillonnant a_2 de la distribution de probabilité $F_{\mathbf{a}}$ et en posant $t_2 = t_1 + a_2$.

- Si $c_1 < t_2$, le clock est avancé de e_1 à $e_2 = c_1$, l'état du service est mis à $B(t_2^+) = 0$ et $Y(t_2^+) = 0$.
- Si $t_2 < c_1$, le clock est avancé de e_1 à $e_2 = t_2$. Puisque le serveur est actif, le nombre de clients en attente devient $Q(t_2^+) = 1$ et le nombre de clients dans le système devient $Y(t_2^+) = 2$.
- Le temps de la troisième arrivée est calculé: $t_3 = t_2 + a_3$ où a_3 est tiré de manière aléatoire.
- et ainsi de suite.

Visualisation de la simulation

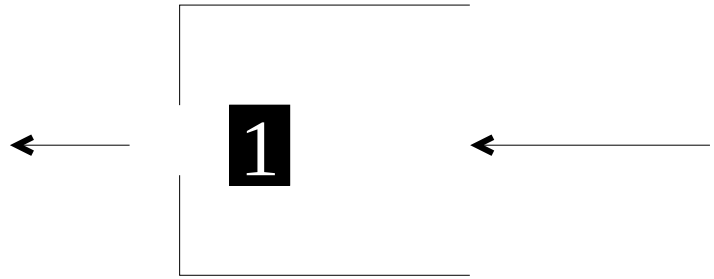


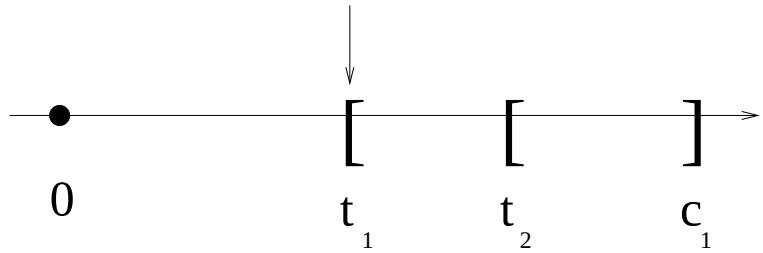


t=0

Q=0, Y=0

B=0





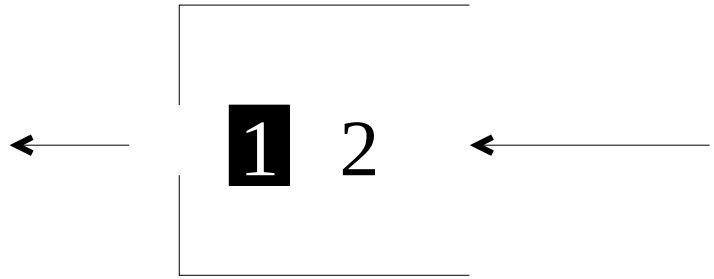
$t=t_1$

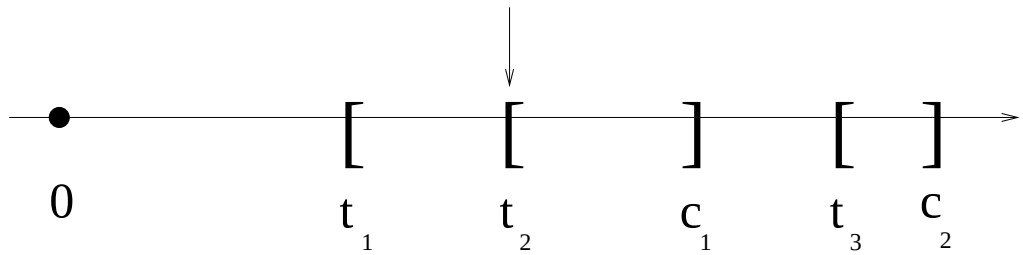
$Y=1, Q=0$

$B=1$

$c_1=t_1+S_1$

$t_2=t_1+A_2$





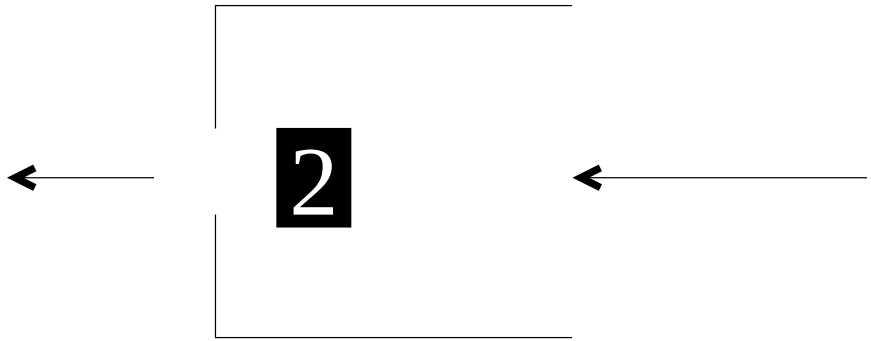
$t=t_2$

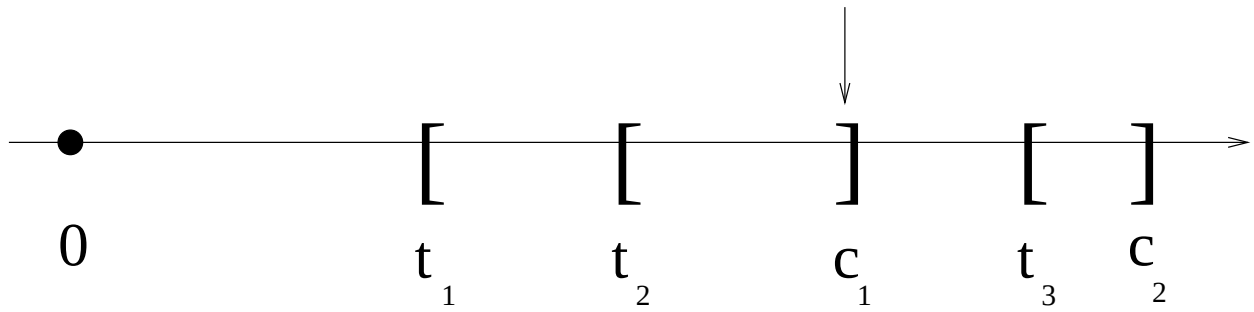
$Y=2, Q=1$

$B=1$

$c_2 = t_2 + S_2 + c_1 - t_2$

$t_3 = t_2 + A_3$

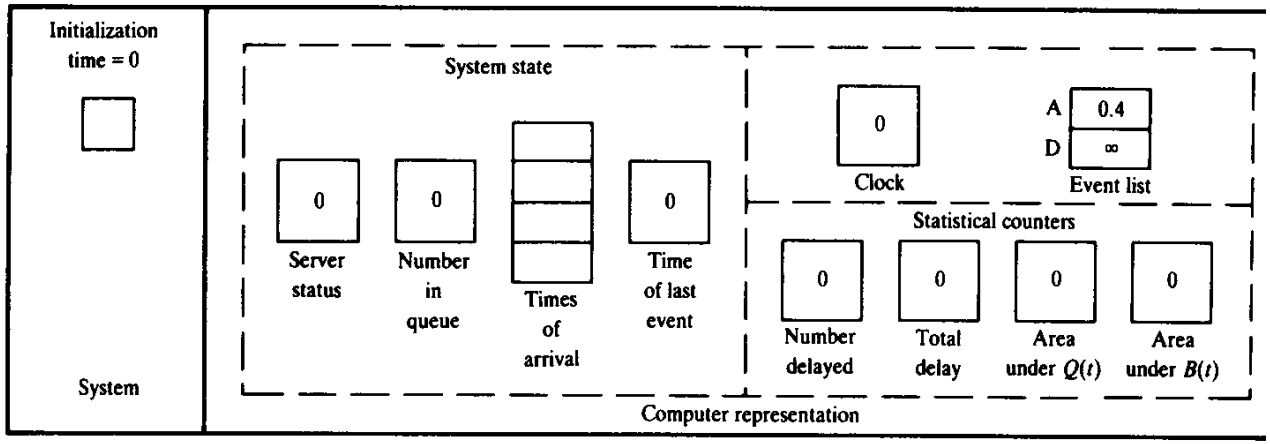




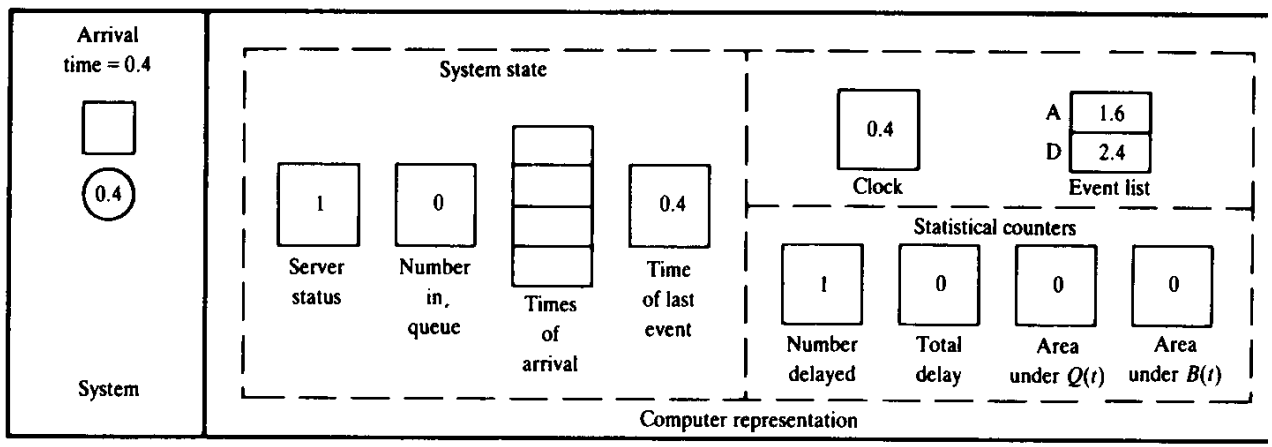
$t=c_1$

$Q=0,\ Y=1$

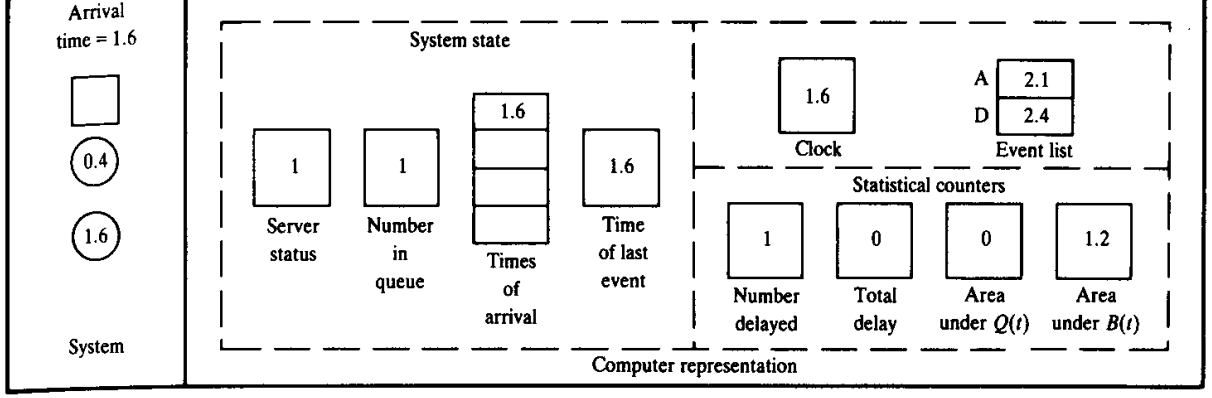
$B=1$


Initialization
time = 0
System
System state
0
Server status
0
Number in queue
Times of arrival
0
Time of last event
0
Clock
A
D
0.4
∞
Event list
Statistical counters
0
Number delayed
0
Total delay
0
Area under $Q(t)$
0
Area under $B(t)$
Computer representation

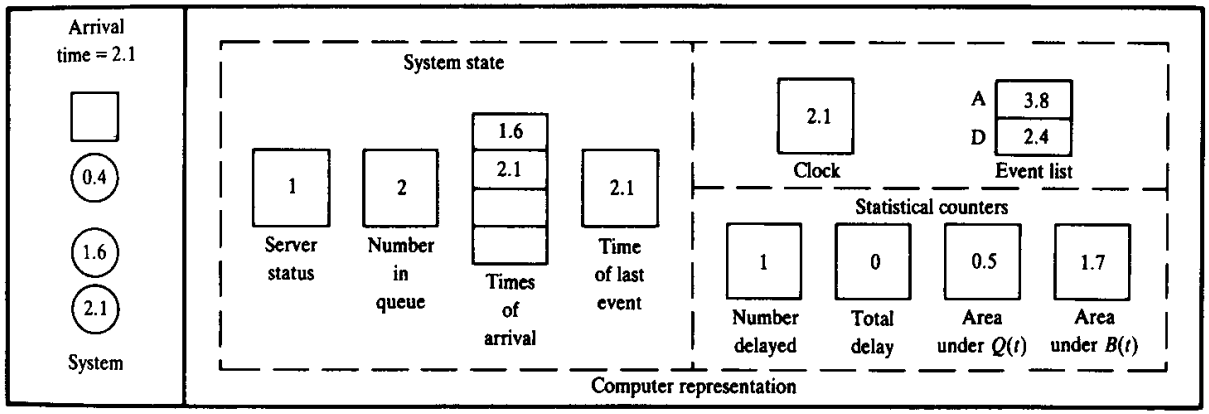
(*a*)


Arrival
time = 0.4
0.4
System
System state
1
Server status
0
Number in queue
Times of arrival
0.4
Time of last event
0.4
Clock
A
D
1.6
2.4
Event list
Statistical counters
1
Number delayed
0
Total delay
0
Area under $Q(t)$
0
Area under $B(t)$
Computer representation

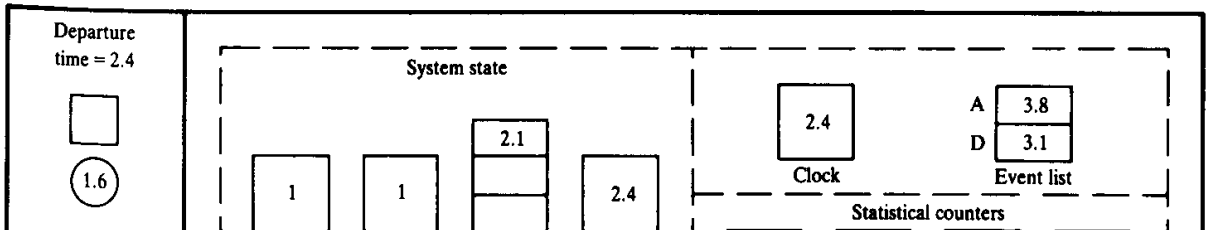
(*b*)

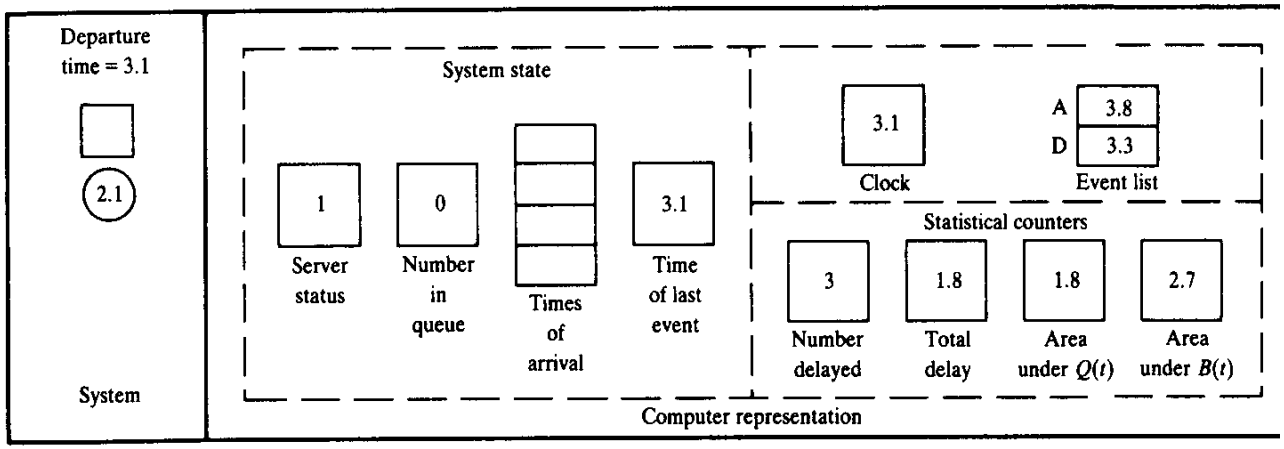

Arrival
time = 1.6
0.4
1.6
System
System state
1
Server
status
1
Number
in
queue
1.6
Times
of
arrival
1.6
Time
of last
event
1.6
Clock
A
D
2.1
2.4
Event list
Statistical counters
1
Number
delayed
0
Total
delay
0
Area
under $Q(t)$
1.2
Area
under $B(t)$
Computer representation

(c)

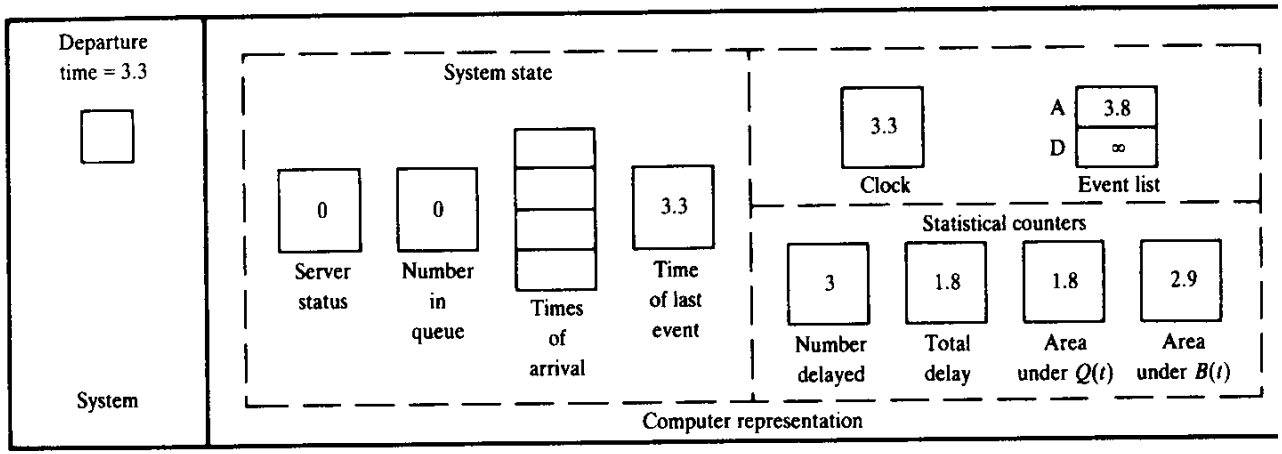

Arrival
time = 2.1
0.4
1.6
2.1
System
System state
1
Server
status
2
Number
in
queue
1.6
2.1
Times
of
arrival
2.1
Time
of last
event
2.1
Clock
A
D
3.8
2.4
Event list
Statistical counters
1
Number
delayed
0
Total
delay
0.5
Area
under $Q(t)$
1.7
Area
under $B(t)$
Computer representation

(d)

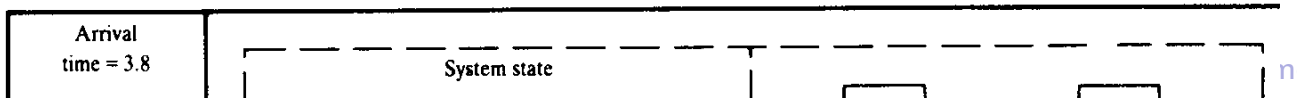

Departure
time = 2.4
1.6
System
System state
1
1
2.1
2.4
2.4
Clock
A
D
3.8
3.1
Event list
Statistical counters

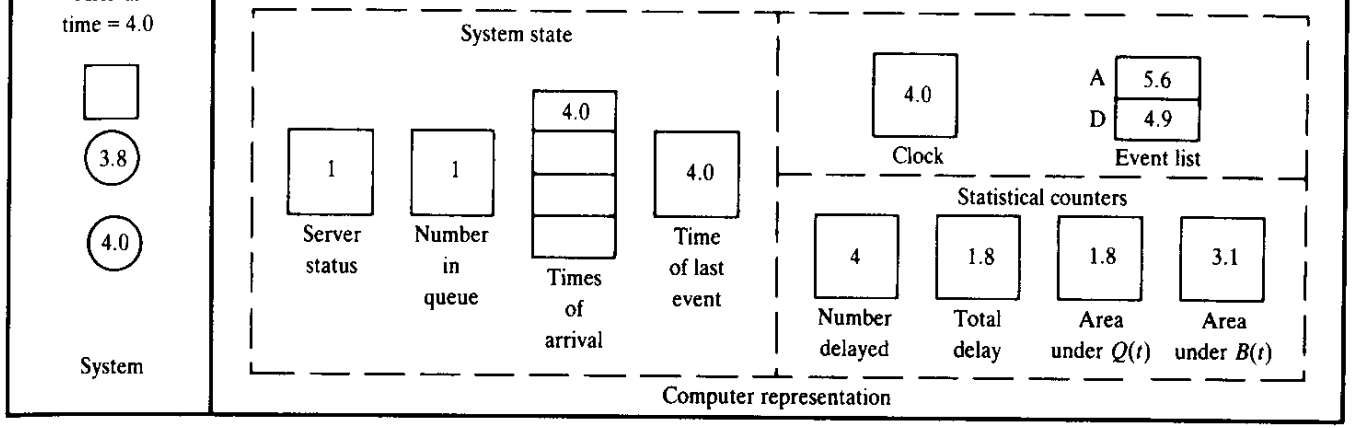

Departure
time = 3.1
2.1
System
System state
1
Server
status
0
Number
in
queue
Times
of
arrival
3.1
Time
of last
event
3.1
Clock
A
D
3.8
3.3
Event list
Statistical counters
3
Number
delayed
1.8
Total
delay
1.8
Area
under $Q(t)$
2.7
Area
under $B(t)$
Computer representation

(*f*)

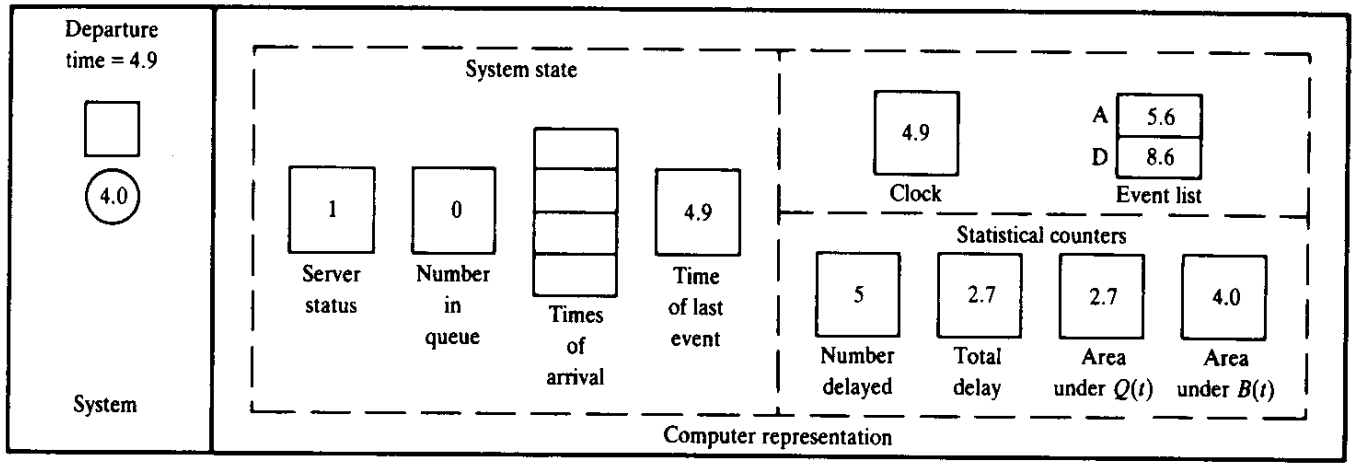

Departure
time = 3.3
System
System state
0
Server
status
0
Number
in
queue
Times
of
arrival
3.3
Time
of last
event
3.3
Clock
A
D
3.8
∞
Event list
Statistical counters
3
Number
delayed
1.8
Total
delay
1.8
Area
under $Q(t)$
2.9
Area
under $B(t)$
Computer representation

(*g*)

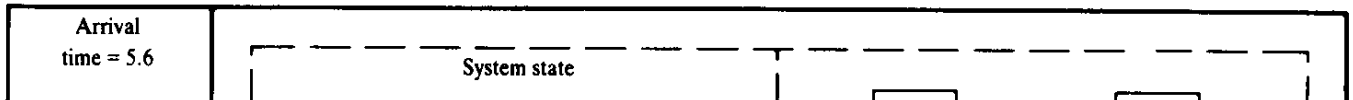

Arrival
time = 3.8
System state


time = 4.0
3.8
4.0
System
System state
1
Server status
1
Number in queue
4.0
Times of arrival
4.0
Time of last event
4.0
Clock
A
5.6
D
4.9
Event list
Statistical counters
4
Number delayed
1.8
Total delay
1.8
Area under $Q(t)$
3.1
Area under $B(t)$
Computer representation

(*i*)


Departure
time = 4.9
4.0
System
System state
1
Server status
0
Number in queue
Times of arrival
4.9
Time of last event
4.9
Clock
A
5.6
D
8.6
Event list
Statistical counters
5
Number delayed
2.7
Total delay
2.7
Area under $Q(t)$
4.0
Area under $B(t)$
Computer representation

(*j*)


Arrival
time = 5.6
System state

Indicateurs de performance

Supposons de considérer une file d'attente qui termine de fonctionner après que n clients ont été servis. Afin de quantifier la performance du système, nous définissons trois quantités:

1. le **temps moyen d'attente** $w_q(n)$ des clients dans la file qui donne le point de vue des utilisateurs.
2. le **nombre moyen de clients en attente** dans la file:

$$L_q(n) = \sum_{i}^{\infty} i p_i$$

   où p_i est la portion de temps de fonctionnement pendant laquelle le nombre de clients en attente est égal à i
3. l'**utilisation moyenne** $\rho(n)$ du service qui correspond à la proportion du temps pendant laquelle le service est actif.

Autres quantités pourraient être utilisés pour caractériser la performance. Aussi, puisque le système est aléatoire, dans le cas plus général, ces quantités peuvent être seulement estimées.

Temps moyen d'attente

Sur base d'une exécution de notre simulateur, nous pouvons estimer les indicateurs. Notons que puisque l'estimation est basée sur des données générées par un processus aléatoire, toutes les quantités estimées sont aussi des variables aléatoires.

- Soient $D_1, \ldots, D_n$ les délais mesurés. Un estimateur intuitif de $w_q(n)$ est calculé par

$$\hat{w}_q(n) = \frac{\sum_{i=1}^{n} D_i}{n}$$

qui est la moyenne arithmétique des n quantités observées.

Nombre moyen de clients en attente

- Soit $L_q(n)$ le nombre moyen des clients en attente.
- Soit $Q(t)$ le nombre des clients en attente à l'instant t, $t \geq 0$.
- La quantité $\hat{L}_q(n)$ peut être écrite

$$\hat{L}_q(n) = \frac{\int_0^{T(n)} Q(t)dt}{T(n)}$$

Utilisation moyenne

- Soit $B(t)$ une fonction qui à l'instant t prend la valeur 0 si le service est inactif et la valeur 1 si le service est actif.
- L'utilisation $\rho(n)$ peut alors être estimée par

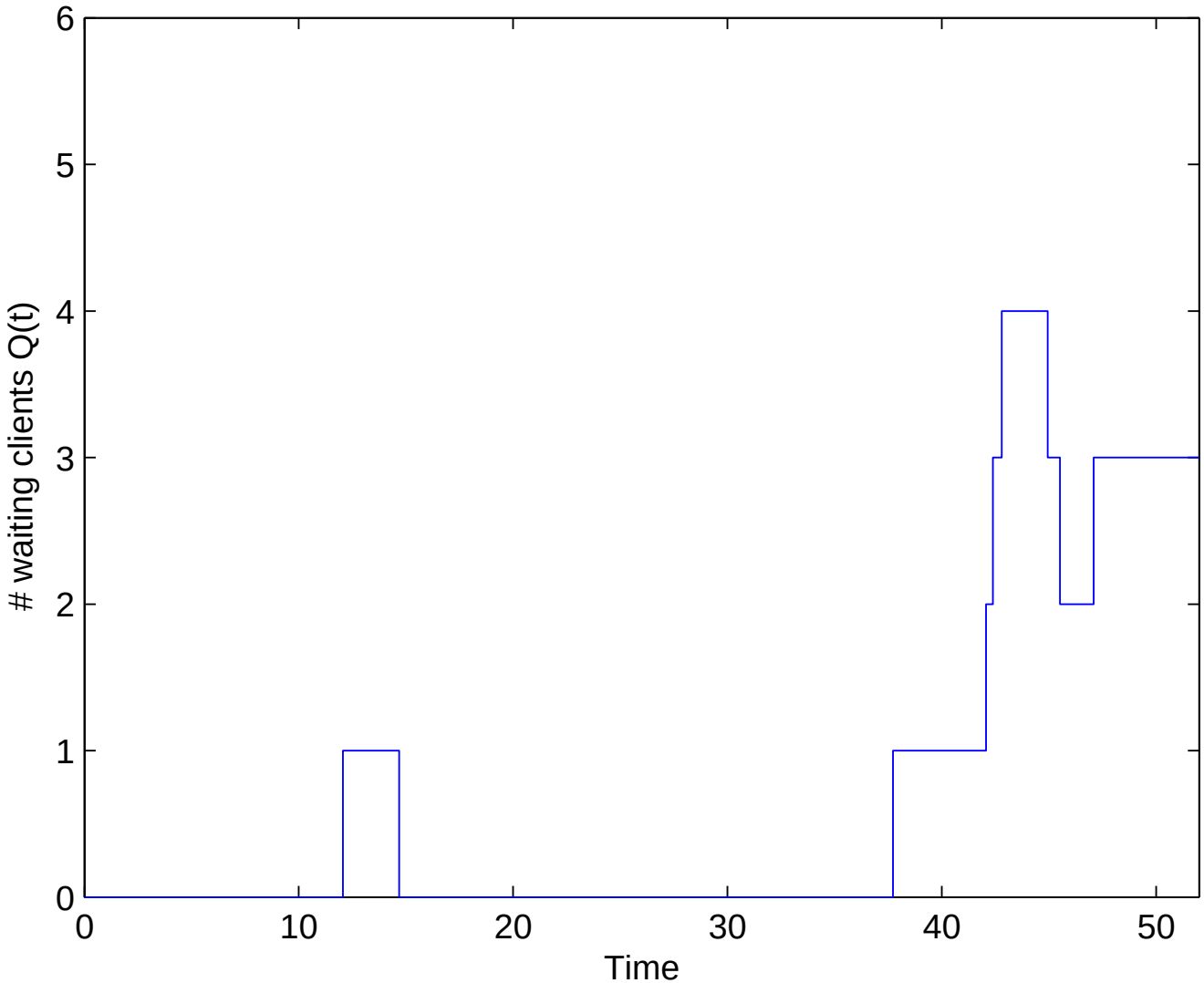
$$\hat{\rho}(n) = \frac{\int_0^{T(n)} B(t)dt}{T(n)}$$

Statistiques

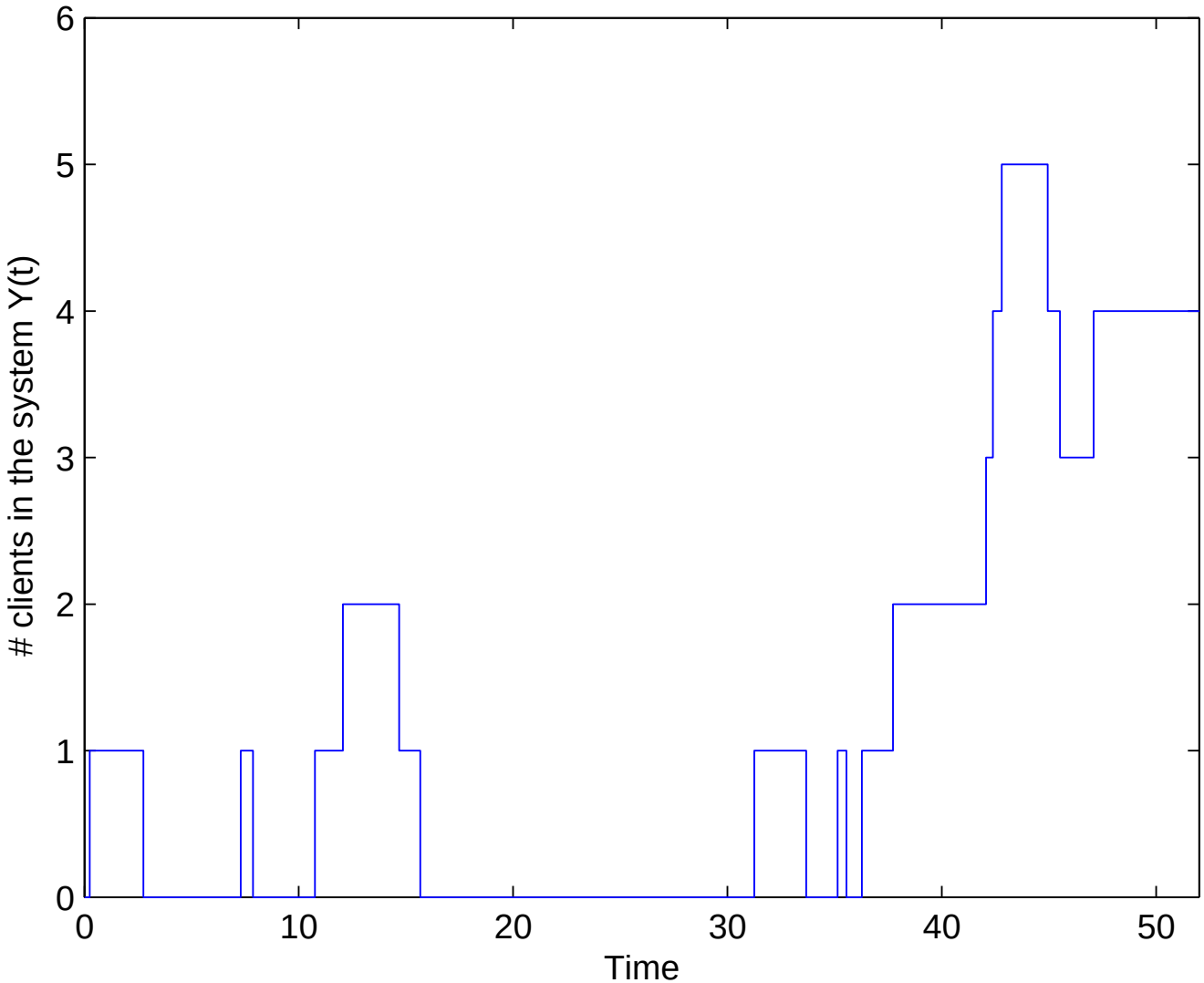
Arrivée	Début service	Service	Départ	Délai	Perm
0.2459	0.2459	2.4974	2.7434	0	2.4974
7.2880	7.2880	0.5754	7.8633	0	0.5754
10.7571	10.7571	3.9212	14.6783	0	3.9212
12.0633	14.6783	0.9837	15.6620	2.6150	3.5987
31.2449	31.2449	2.4272	33.6721	0	2.4272
35.1408	35.1408	0.4071	35.5479	0	0.4071
36.2623	36.2623	8.6788	44.9412	0	8.6788
37.7216	44.9412	0.5622	45.5034	7.2195	7.7817
42.0588	45.5034	14.2459	59.7493	3.4446	17.6905
42.3795	59.7493	9.8703	69.6196	17.3698	27.2402
42.7966	69.6196	7.9785	77.5982	26.8231	34.8016
47.0799	77.5982	2.5226	80.1208	30.5182	33.0409
52.0879	80.1208	6.5063	86.6271	28.0329	34.5392

Données générées par la simulateur en Octave

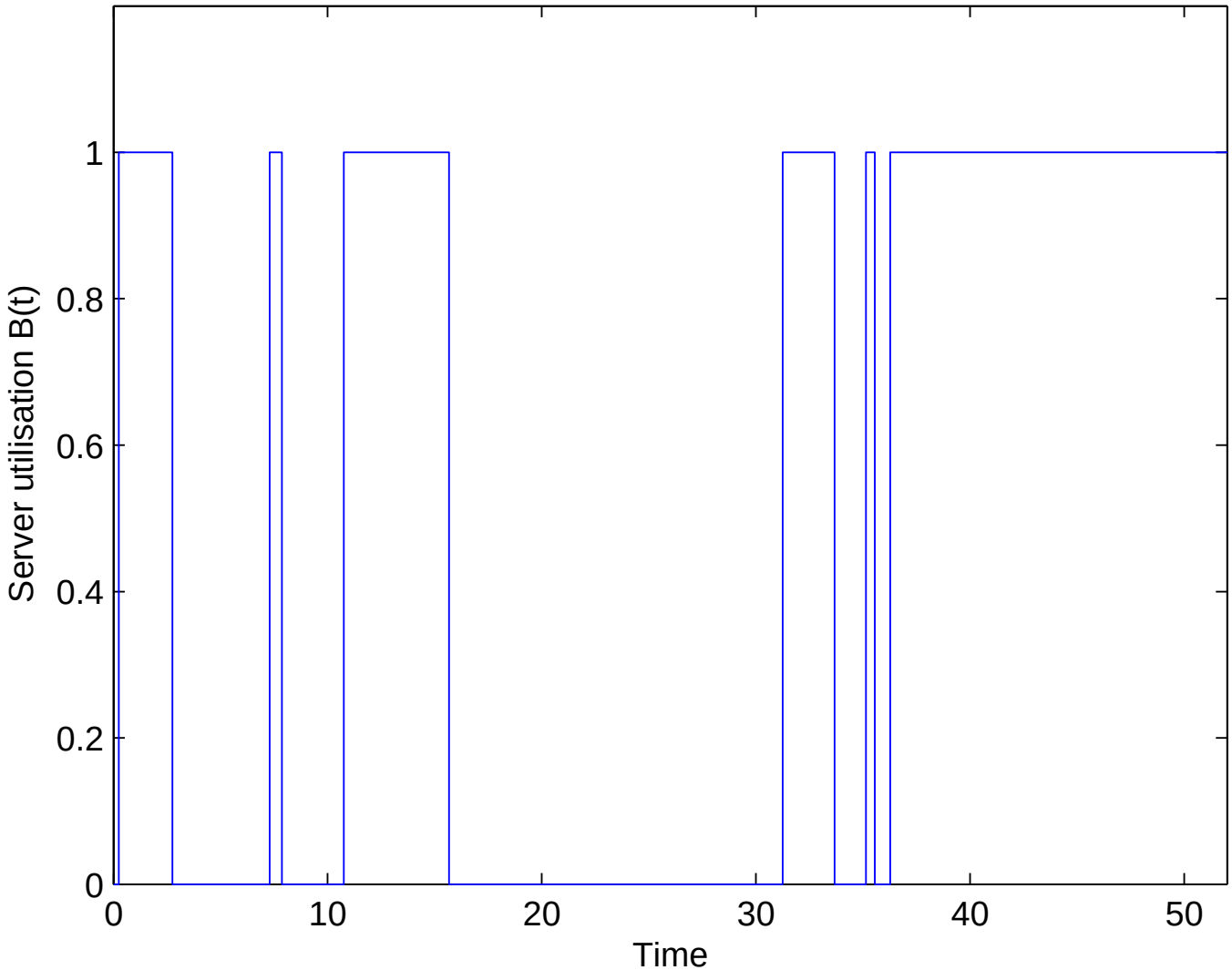
$Q(t)$: clients en attente



$Y(t)$: clients dans le système



Fonction $B(t)$



Résultats analytiques $M/M/1$

- Dans le cas d'une file d'attente $M/M/1$ il est possible calculer les valeurs des indicateurs de performance sans passer par la simulation.
- Si le temps entre deux arrivées suit une distribution exponentielle ayant moyenne $1/\lambda$ et le temps de service suit une distribution exponentielle ayant moyenne $1/\mu$, avec $\mu > \lambda$ il est possible montrer que pour $n \to \infty$ la file d'attente atteigne un état d'équilibre statistique, où

$$p(Q(t) = i) = p_i$$

c.-à-d. la probabilité qu'il y ait i clients en attente est indépendante de t et

$$\lim_{n\to\infty} \hat{L}_q(n) = L_q, \quad \lim_{n\to\infty} \hat{\rho}(n) = \rho \quad \lim_{n\to\infty} \hat{w}_q(n) = w_q$$

Résultats analytiques $M/M/1$

Il est possible montrer aussi que

- utilisation moyenne $\rho = \lambda/\mu$
- nombre moyen de clients dans le système $L = \frac{\rho}{1-\rho}$
- nombre moyen de clients dans la file $L_q = \frac{\rho^2}{1-\rho}$
- le temps moyen d'attente dans le système $w = \frac{1}{\mu-\lambda}$.
- le temps moyen d'attente dans la file $w_q = \frac{\rho}{(1-\rho)\mu}$
- densité de probabilité du temps d'attente dans la file: $p_q(t) = \rho(\mu - \lambda)e^{-t(\mu-\lambda)}$
- densité de probabilité du temps dans le système: $p(t) = (\mu - \lambda)e^{-t(\mu-\lambda)}$

Simulateurs à événements discrets

- La programmation (et le test) d'un simulateur à événements discrets est souvent une tâche difficile parce que ceci est un exemple de programmation en parallèle où il faut simuler plusieurs activités concurrentes.
- Pour cette raison, dans la communauté informatique, plusieurs langages de simulation ont été proposés afin de permettre au programmeur de mieux gérer la complexité du problème.
- Un exemple est le langage SIMULA, inventé en 1960, qui a été parmi les premiers langages à utiliser la notion d'orienté objet.
- La tendance actuelle est toutefois de créer des librairies pour la simulation qui peuvent être exploitées à l'intérieur de programmes codés avec des langages communs (par exemple C++).
- Il est donc plus instructif de parler de paradigmes de programmation pour la simulation plutôt que des langages de programmation ad-hoc.

Event-based vs. *process-based*

Deux approches sont communément utilisées pour programmer un simulateur d'un système à événements discrets:

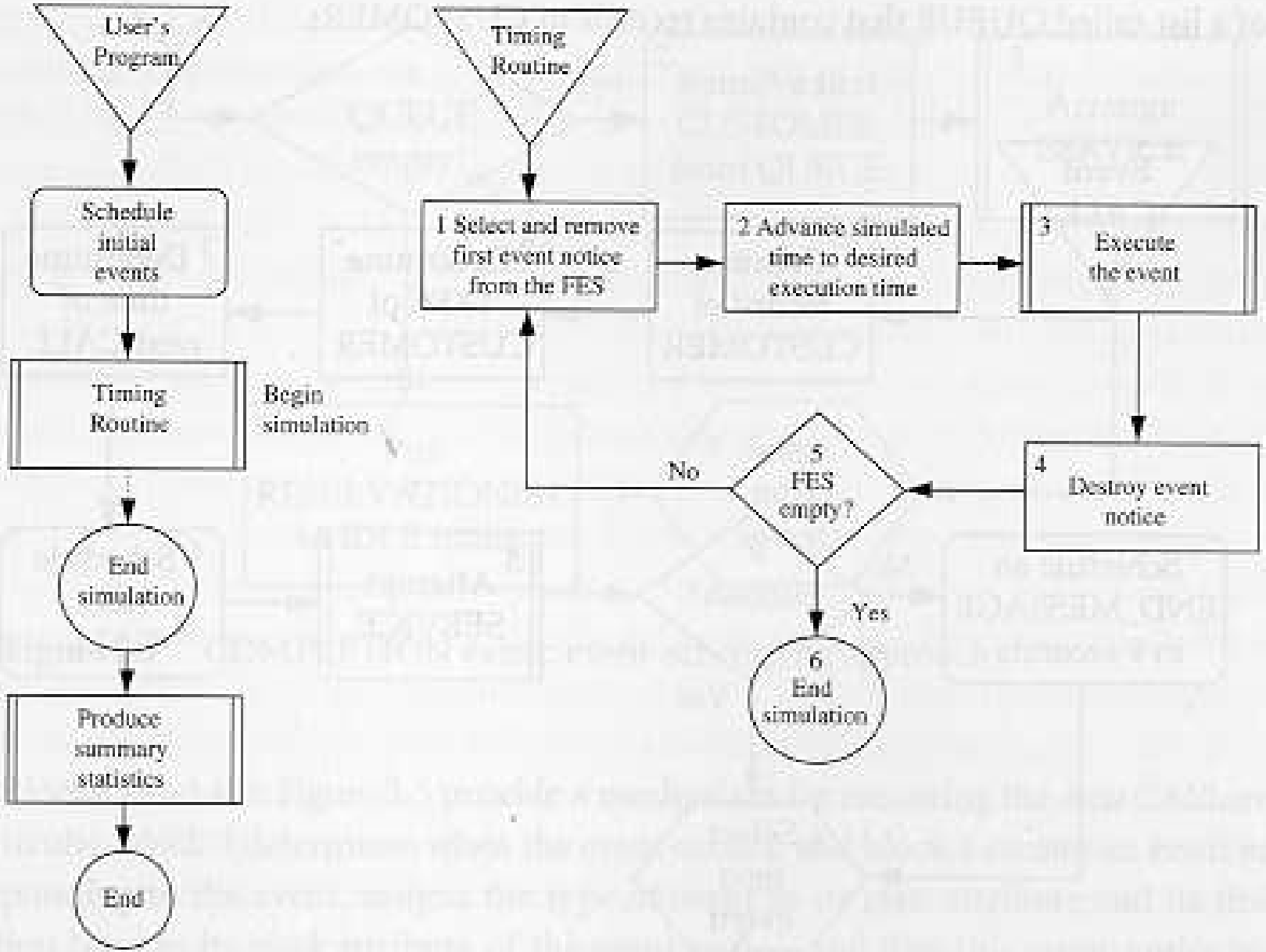
Event-based: l'approche se focalise sur la suite d'événements (par exemple les arrivées et les départs) indépendamment des entités concernées

Process-based: l'approche met l'accent sur les différentes entités (par exemple client, serveur) et sur leur interaction.

Approche event-based

- Cette approche demande la définition d'une liste qui contient l'ensemble des événements futurs à venir.
- L'approche est implémentée par une boucle qui
  1. sélectionne et enlève le prochain événement dans la liste
  2. donne le contrôle à une sous-routine qui exécute la liste des modifications liées à l'événement.
  3. éventuellement, mets à jour la liste avec des événements futurs
- Cette approche peut être implémentée par quelconque langage impératif (par exemple Octave) où il est possible définir une liste et qui contient des générateurs de nombre aléatoires.
- Toutefois, surtout s'ils existent plusieurs types d'événements, l'implémentation pourrait être rendue plus aisée par l'existence d'une structure de données du genre *priority queue*.

Approche *event-based*



Avantages de l'approche event-based

- Facilité d'implémentation.
- Possibilité d'utiliser langages *general-purpose*.
- Vitesse d'exécution.
- Flexibilité. Il est par exemple facile d'implémenter un système où un événement en déclenche deux autres.

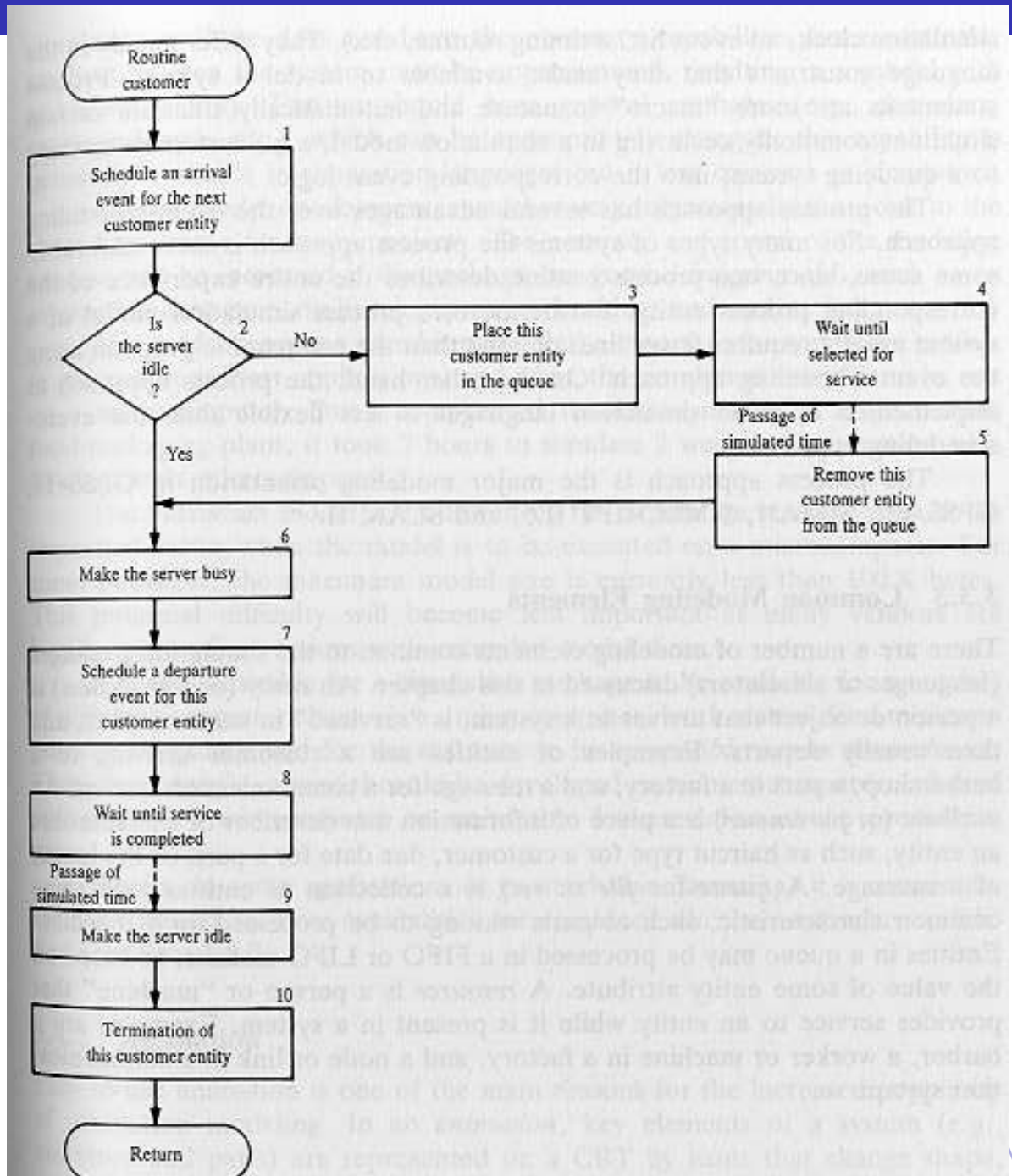
Approche *process-based*

- Cette approche associe un processus à chaque entité et modélise le système comme un ensemble d'entités interagissant.
- Chaque processus modélise l'avancement du temps.
- Un exemple de processus est le processus *client* qui arrive, se met en attente, est réveillé par la mise à disposition d'un guichet, attend le temps d'exécution de l'opération et puis s'en va du système.
- Le paradigme orienté-objet est souvent utilisé pour modéliser une représentation process-based du système.

Approche *process-based*

- La représentation process-based est souvent plus convivial et plus compacte, puisque elle permet de décomposer un problème de simulation en termes d'entité ou d'agents.
- Ce style de programmation est rendu possible par le fait que plusieurs systèmes d'exploitation mettent en oeuvre la notion de *threads* (ou processus légers) (par exemple *pthreads* en Unix, *Java threads*, *Windows threads*). Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle.
- C++SIM et SimPy sont deux packages pour la simulation process-based.

Approche *process-based*



simpy

simpy est un package Python qui permet de créer très facilement des modèles de systèmes à événements discrets.
simpy mets à disposition du programmeur plusieurs fonctionnalités:

- **Process:** un *g*enerator python est utilisé pour simuler une entité qui évolue avec le temps, par exemple un job qui doit être exécuté par une machine ou un bateau qui doit être déchargé à l'intérieur d'un terminal container.
- **Resource**: cette classe représente une ressource FIFO qui doit être partagée et pour laquelle on peut se mettre en attente (par exemple une machine, un guichet, une grue).
- **Environment:** tous les processus vivent dans un environnement et ils interagissent avec l'environnement et les autres processus par le biais d'*events*

et les fonctionnalités suivantes:

- **process():** il rend un processus exécutable après avoir été créé.
- **yield:** produit un événement, passe le contrôle au simulateur et il se met en attente
- **timeout:** type d'événement qui est activé après un certain intervalle de temps
- **request:** demande une ressource et se met en attente
- **release:** relâche une ressource
- **interrupt:** interrompt l'action
- **run**: démarre la simulation

Notons que l'exécution d'un processus est arrêtée seulement lors d'une commande **yield**.

Example: gestion cinéma

Objectif: optimiser la quantité de personnel dans un cinéma afin de minimiser le temps d'attente des clients, tout en respectant des contraintes budgétaires:
`https://realpython.com/lessons/defining-simulation/`
Etape d'un client dans un cinéma:

1. il arrive au cinéma
2. il achète un ticket
3. il fait valider son ticket
4. parfois, il achète à manger
5. il se rend dans la salle

Processus

```
def purchase_ticket(self, moviegoer):
        lam=0.5
        yield self.env.timeout(random.expovariate(lam))

    def check_ticket(self, moviegoer):
        yield self.env.timeout(3 / 60)

    def sell_food(self, moviegoer):
        lam=0.1
        yield self.env.timeout(random.expovariate(lam))
```

La simulation demande une estimation (de la distribution) des temps d'exécution des différentes opérations.

Ressources

Les processus demandent d'abord l'accès à une quantité limitée de ressources (p.ex. guichetier, vendeur de pop corn).

```
self.cashier = simpy.Resource(env, num_cashiers)
self.server = simpy.Resource(env, num_servers)
self.usher = simpy.Resource(env, num_ushers)
```

Agent

Fonction go_to_movies(moviegoer, theater):

```
 with theater.cashier.request() as request:
        # moviegoer generates a request to use a cashier.
        print('%7.4f : person %s waiting for cashier' % (env.now, moviegoer))
        yield request

        print('%7.4f : person %s got cashier' % (env.now, moviegoer))
        yield env.process(theater.purchase_ticket(moviegoer))
        print('%7.4f : person %s has purchased the ticket ' % (env.now, moviegoer))

with theater.usher.request() as request:
        yield request
        print('%7.4f : person %s got usher' % (env.now, moviegoer))
        yield env.process(theater.check_ticket(moviegoer))
        print('%7.4f : person %s has checked the ticket ' % (env.now, moviegoer))

 if random.choice([True, False]):
        with theater.server.request() as request:
            yield request
            print('%7.4f : person %s is being served food' % (env.now, moviegoer))
            yield env.process(theater.sell_food(moviegoer))
            print('%7.4f : person %s got food ' % (env.now, moviegoer))
    # Moviegoer heads into the theater
    delay=env.now - arrival_time
    wait_times.append(delay)
```

Rappels Python: générateur (mot clé yield), itérateur

Scénario simulation

3 clients déjà à l'ouverture. Les autres arrivent suivant une loi de Poisson (boucle infinie).

```
def run_theater(env, num_cashiers, num_servers, num_ushers):
    theater = Theater(env, num_cashiers, num_servers, num_ushers)

    for moviegoer in range(3):
        env.process(go_to_movies( moviegoer, theater))

    while True:
     lam=2 ## avg. frequency (2 persons per minute)
        yield env.timeout(random.expovariate(lam))  # Wait a bit before generating a new person
        moviegoer += 1
        env.process(go_to_movies( moviegoer, theater))
```

Exécution simulation

La simulation s'arrête après 10 minutes. On aurait pu choisir d'autres stratégies d'arrêt.

```
def main():
    # Setup
    random.seed(42)
    num_cashiers=1
    num_servers=1
    num_ushers = 3

    # Run the simulation
    env = simpy.Environment()
    env.process(run_theater(env, num_cashiers, num_servers, num_ushers)
    env.run(until=10)
```

Analyse statistique

```
average_wait = statistics.mean(wait_times)

print(
        "Running simulation...",
        f"\nThe average wait time is {average_wait} minutes ...",
    )
```

On peut bien évidemment imaginer des analyses/affichages plus détaillées.

Bonus: étude de trafic par simulation

Simuler en `simpy` un carrefour avec un feu entre 2 routes, une principale et une secondaire.
Le trafic dans la route principale a un débit de 1 voiture par seconde. Le trafic dans la route secondaire a un débit de 0.1 voitures par seconde. Le feu est vert pour la route primaire pendant 50 secondes par minute. Les voitures emploient 0.1 secondes pour traverser le carrefour.
L'étudiant devra

- Fournir le code dans un notebook python.
- Estimer le temps moyen d'attente dans la route primaire et secondaire.
- Estimer l'impact sur le temps moyen d'attente si la durée du feu vert est identique au rouge.

Les mots de la fin

- Tous les modèles vus dans le cours demandent une connaissance à priori détaillée du problème à modéliser et simuler.
- Connaissance humaine est très précieuse et elle doit toujours être employée si disponible et formalisable de manière mathématique.
- Ce type de connaissance est particulièrement utile dans les sciences appliquées traditionnelles (physique, chimique, économie)
- Quid des domaines/problèmes moins conventionnels pour lesquels la connaissance peut être indisponible ou difficile à formaliser (sociologie, génomique, sécurité) ?
- Solution: modélisation à partir des données, statistique, apprentissage automatique,...
- Au revoir au cours INFOF422 "Statistical foundations of machine learning"...