HMIN105M - projet
Système de réservation distribué

Synthèse du projet :

« L'idée de ce projet s'inspire d'un système de réservation de ressources de calcul et/ou de stockage sur une plateforme de grille ou de cloud. Il a pour objectif de permettre à des clients de louer des puissances de calcul ou des espaces de stockage distants répondant à des besoins spécifiques (exemple : exécuter une simulation scientifique sur une architecture distribuée de processeurs et de mémoire pour stocker (le temps de la location) les données traitées et produites).
Dans le système à mettre en œuvre, un client aura la possibilité de louer des ressources, soit en mode exclusif (les ressources louées sont utilisées par un seul client pendant toute la durée de la réservation), soit en mode partagé (les ressources louées peuvent être utilisées en même temps par plusieurs clients). [...] »

Architecture de l'application

Objets IPC utilisés :

Afin de réaliser ce projet, nous avons choisi de mettre en place 3 types d'objets IPC : des segments de mémoire partagée, des tableaux de sémaphore et un verrou.

Il y a précisément 2 segments de mémoire partagée :

- Le premier segment permet de connaître la quantité (initiale, libre et utilisée) de *CPU* et de *stockage* pour chacun des *sites* de notre programme. Il permet également de connaître la quantité de *CPU* et de *stockage* possédé (en exclusif comme en partagé) par chacun des *clients* d'un *site* donné.

- Le deuxième segment permet simplement d'enregistrer la demande de ressource d'un *client* ou bien le fait qu'un *client* rende des ressources. Cela permet d'informer tous les autres clients et joue donc le rôle de notification.

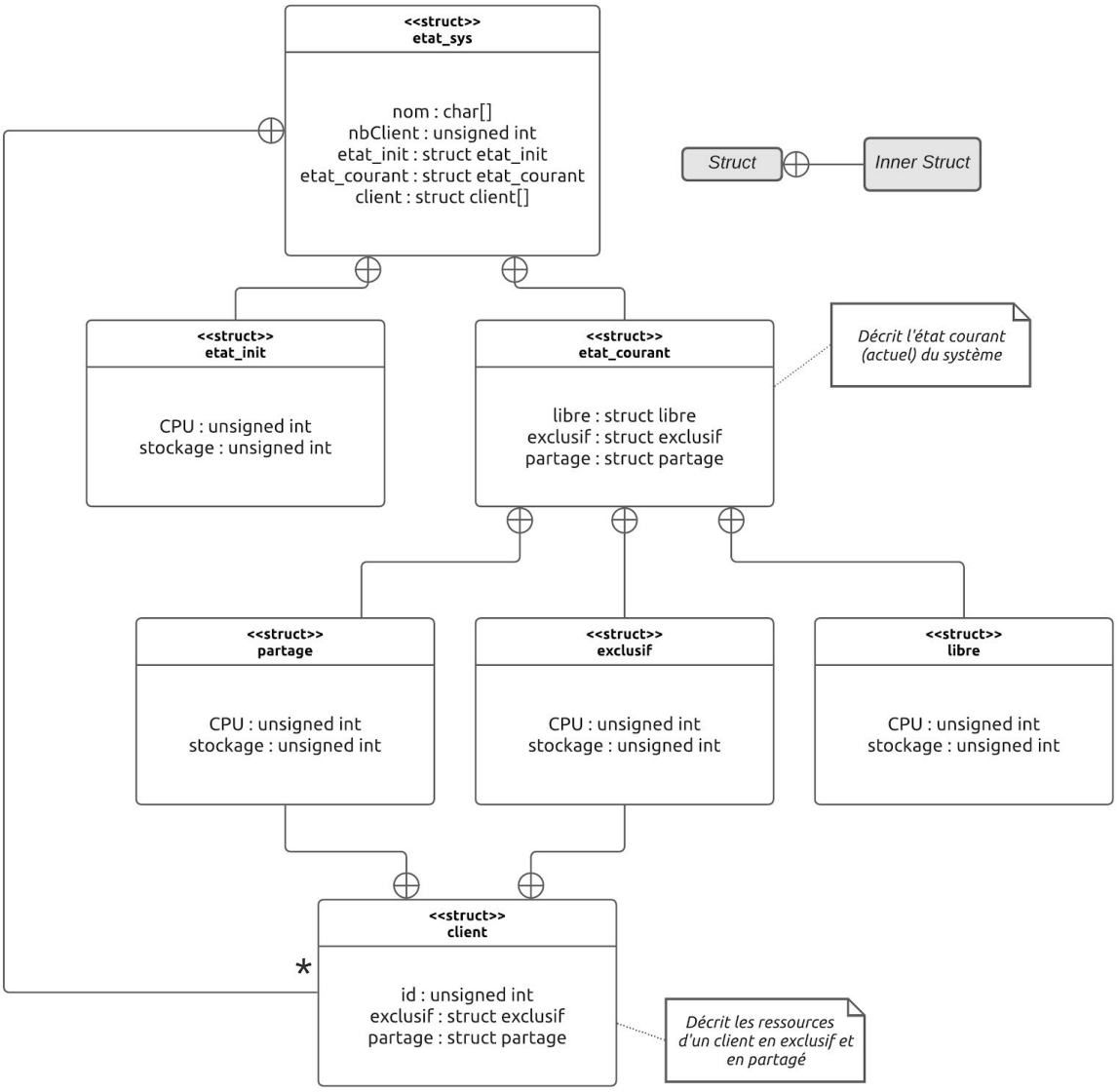


Figure 1 - Schéma UML de la structure du 1er segment de mémoire partagé

Ce premier segment permet donc d'obtenir toutes les informations concernant l'état du système. Sa taille est égale au nombre de sites proposant des ressources dans notre système. Il se manipule donc comme un tableau où le premier élément de ce tableau correspond au premier site, le deuxième élément de ce tableau au deuxième site etc… C'est ainsi que pour chaque site de notre segment nous avons accès aux informations suivantes : *Nom du site, nombre de client ayant des ressources sur site donné, état initial (nombre de CPU et quantité de stockage disponible initialement) sur le site, état courant (nombre de CPU et quantité de stockage disponible en temps réel) sur le site, information sur tous les clients ayant des ressources sur le site.*

Le deuxième segment a pour fonction de notifier une demande ou une libération de ressource.

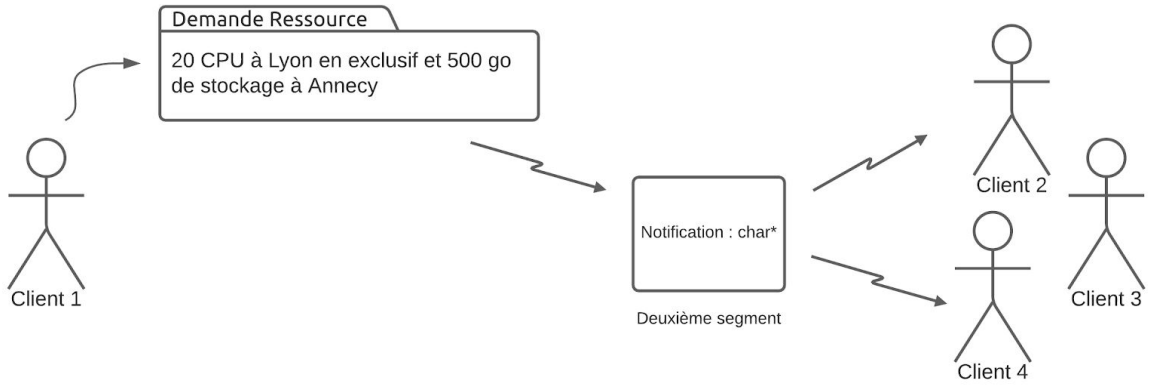


Figure 2 - Schéma indiquant le rôle du deuxième segment

Comme nous pouvons le constater sur la *figure 2* le deuxième segment permet de notifier aux autres clients le fait qu'un client ait effectué une demande de ressource ou ait rendu des ressources. Ce segment est donc de taille 1 et contient une chaîne de caractère. La notification s'effectue au travers de *Threads* propres à chaque client.

En complément des ces 2 segments de mémoire partagés, nous disposons également de 2 tableaux de sémaphores.
Le premier tableau de sémaphore est de taille $2 \times nombre\ de\ site$. Chaque sémaphore du tableau constitue une quantité de ressource disponible. La première moitié du tableau décrit la quantité de CPU disponible et la deuxième moitié du tableau décrit la quantité de stockage disponible.

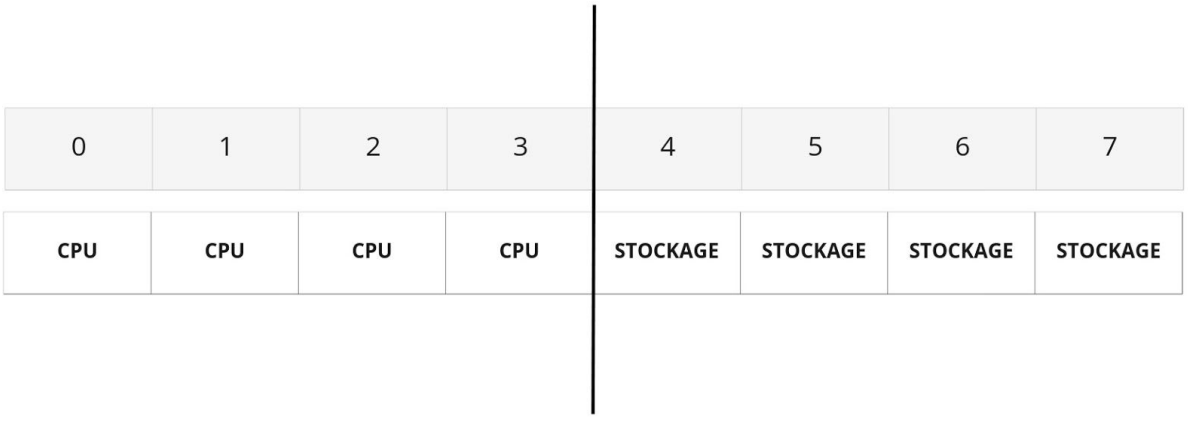


Figure 3 - Schéma indiquant l'organisation de notre premier tableau de sémaphore

Dans l'exemple de la *Figure 3*, nous considérons 4 sites dans notre système. Le premier site est associé aux sémaphores n°0 et n°4, le deuxième site est associé aux sémaphores n°1 et n°5, le troisième site est associé aux sémaphores n°2 et n°6 et enfin le quatrième et dernier site est associé aux sémaphores n°3 et n°7. Chaque fois qu'un client effectue une demande en ressource, la valeur des sémaphores associée aux sites sur lesquels la demande est effectuée est affectée.

Le deuxième tableau de sémaphore quant à lui a une taille fixe qui est de 3.
Son rôle est d'assurer l'unicité de présence des différents processus *clients* au sein de nos deux segments de mémoire partagé. Autrement dit, il joue le rôle de verrou sur nos deux segments. Par ailleurs, il permet également d'être l'objet sur lequel on peut "déclencher" une notification au travers d'un point de rendez-vous sur tous les threads.

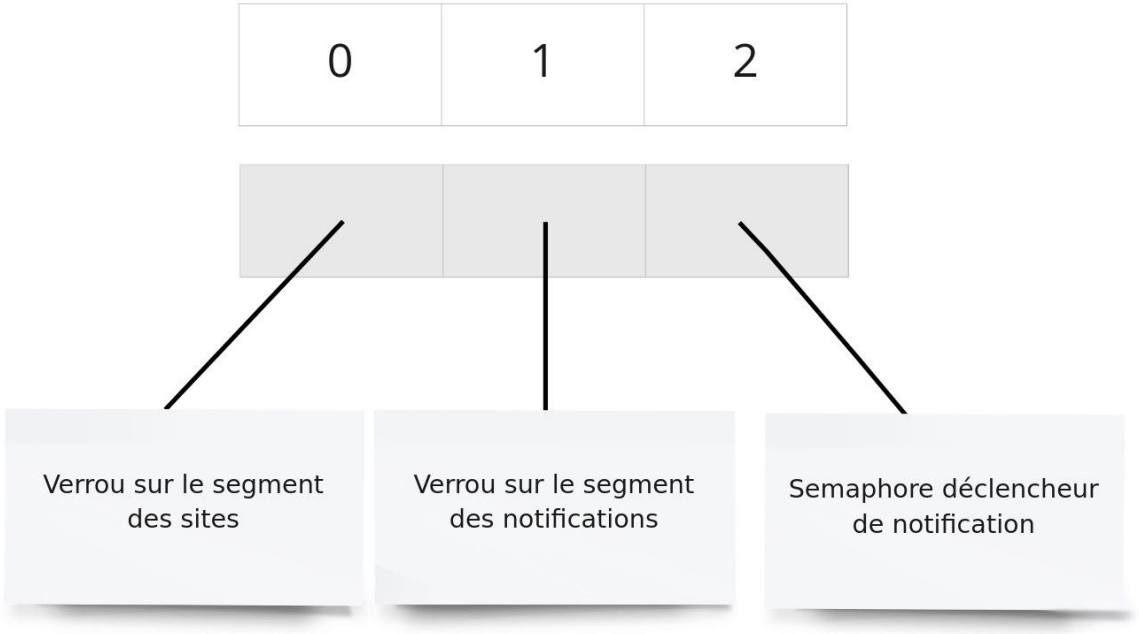


Figure 4 - Schéma descriptif de notre deuxième tableau de sémaphore

Notre troisième et dernier objet IPC est un mutex permettant au mieux d'éviter un affichage saccadé lorsqu'un client reçoit une notification.

Développement de certains points clés du programme :

1. Relation entre notre premier segment de mémoire partagé et notre premier tableau de sémaphore.

Le segment de mémoire partagé descriptif des différents sites de notre programme est initialisé par le serveur avec les quantités de ressources disponibles de chacun des sites. Le tableau de sémaphore associé au segment est également initialisé avec les valeurs des ressources disponibles de chacun des sites présents dans le segment. Chaque fois qu'un client effectue une demande en ressource, sa demande est comparé avec les valeurs en ressources disponibles présentes dans le segment et peut être modifié pour garder une cohérence de l'état du système.

exemple :

Supposons qu'il y ait un unique site : Rennes. Un client C1 utilise déjà des ressources sur le site de Rennes : 20 CPU en mode partagé. Un client C2 effectue une demande de 30 CPU en mode partagé également sur le site de Rennes. Une fois sa demande effectuée, celle-ci est comparée avec la quantité de CPU utilisée en mode partagé sur le site de Rennes. Si la demande est satisfiable alors le client obtient les 30 CPU en mode partagé et une opération P dont la valeur est 10 est réalisée sur le sémaphore descriptif du CPU correspondant.

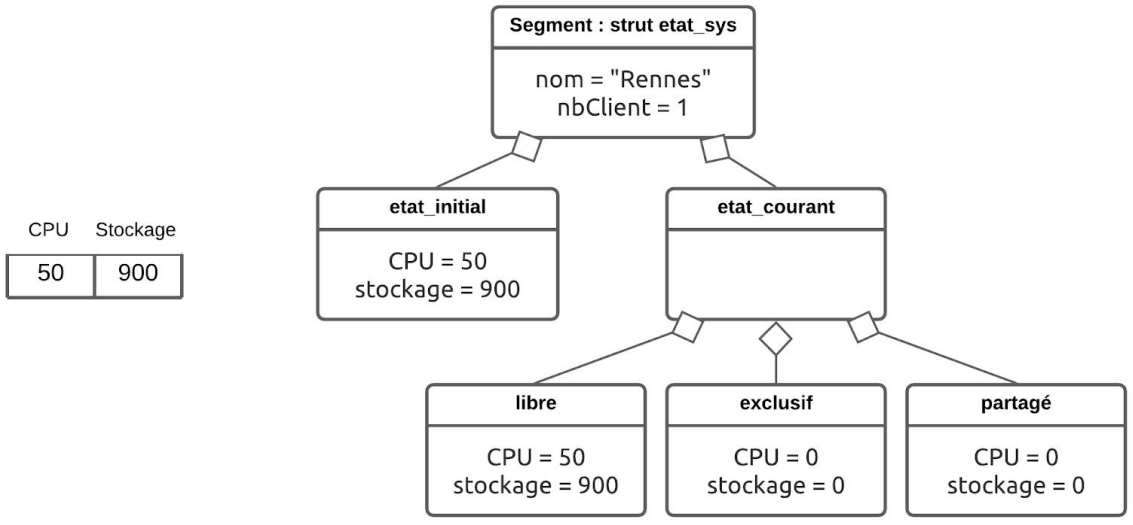


Figure 5 - Schéma descriptif de l'état du système (à droite) et de son tableau de sémaphore associé (à gauche)

Comme nous pouvons le constater sur la *Figure 5*, lorsqu'aucun client n'a encore effectué de réservation la valeur des sémaphores présente dans le tableau (à gauche) est identique à la valeur de l'état initial des ressources du système et par la même occasion, à la valeur des ressources libre (et donc disponible) du système.

Maintenant observons la modification du système une fois que le client C1 a effectué sa demande en CPU :

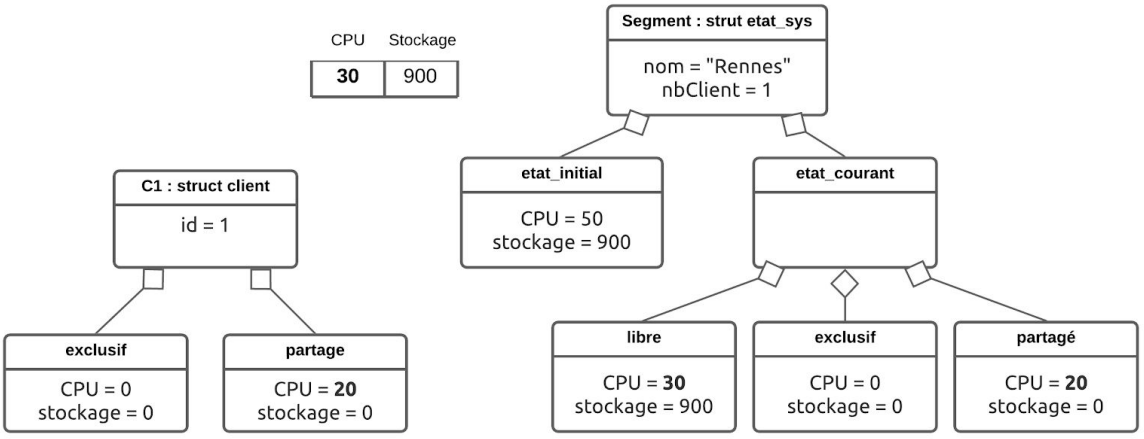
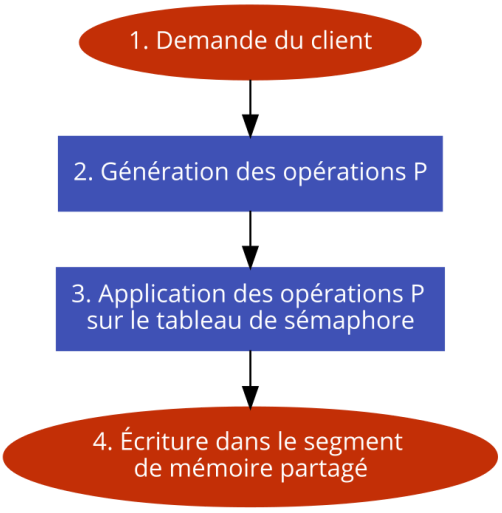


Figure 6 - Modification de l'état du système et du tableau de sémaphore associé

Une fois que le client C1 a effectué sa demande de 20 CPU en partagé, dans l'ordre a lieu les actions suivantes :

L'étape 1 constitue la demande du client, celle-ci est tout le temps satisfaite sauf si le client rentre des saisies erronées ou non satisfiable (cf : situation d'autoblocage).



L'étape 2 permet la génération des opérations P correspondantes que l'on s'apprête à appliquer sur le tableau de sémaphore. Ces opérations sont réalisées de la manière suivante :

- Si la demande est *exclusive* alors on effectue simplement une opération P dont la valeur est : *la demande du client*
- Si la demande est *partagée* alors 2 cas possible : la demande du client est inférieure ou égale à la valeur de la ressource partagée dans le segment, dans ce cas l'opération P est annulée. Maintenant si la demande du client est strictement supérieure à la valeur de la ressource partagée dans le segment alors l'opération P aura la valeur suivante : (*demande du client - valeur de la ressource partagée dans le segment)*.

L'étape 3 est en réalité l'application de l'opération *semop* sur notre tableau de *sembuf* (tableau constituant l'ensemble des demandes du client).

L'étape 4 désigne la modification du segment en fonction de la demande du client et l'ajout du client dans un tableau client associé au site sur lequel il a effectué sa location de ressource.

Maintenant voyons la modification de l'état du système lorsqu'un deuxième client effectue une réservation de *CPU* en mode partagé :

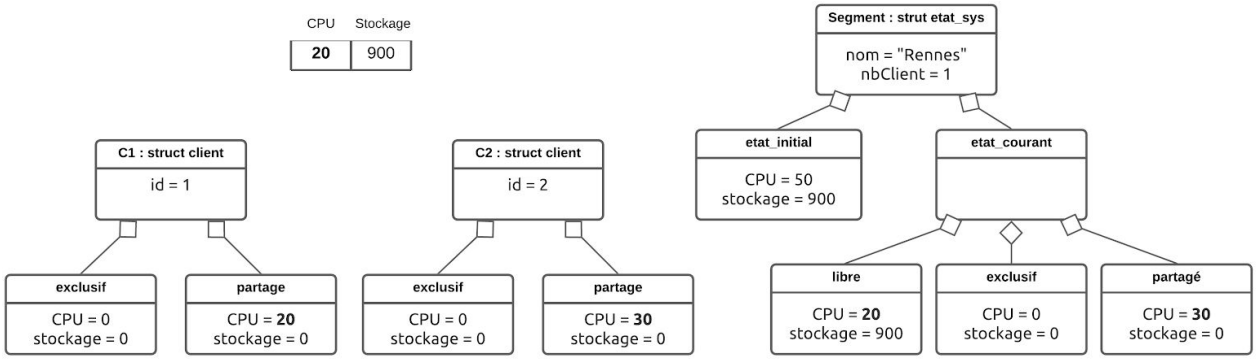


Figure 7 - Le client C2 effectue une réservation de CPU en partagé

Un client C1 possédait déjà 20 CPU en partagé sur le site de Rennes. Un client C2 a effectué une demande de 30 CPU en partagé sur le site de Rennes. Sa demande est satisfaite. Une opération P est appliquée sur le sémaphore correspondant au site de Rennes. La valeur de cette opération est : (*demande du client - CPU en partage sur le site de Rennes)* (car comme dit précédemment, la valeur de la demande du client est strictement supérieure à la valeur de CPU en partagé utilisé sur le site de Rennes). On obtient donc *30-20 = 10* et donc de fait, la valeur sur sémaphore correspondant au CPU de Rennes passe de *30* à *20*. Le segment est ensuite mis à jour.

2. Le rôle du deuxième tableau de sémaphore : exclusion et point de rendez-vous.

Comme dit précédemment, le deuxième tableau de sémaphore est de taille 3. Chaque sémaphore du tableau joue un rôle particulier. Le premier sémaphore est un verrou sur le segment de mémoire partagé correspondant à l'état du système. Il permet d'assurer le fait qu'un unique processus à la fois accède au segment. Le deuxième sémaphore quant à lui est un verrou sur le segment de mémoire partagé correspondant à la notification. Il permet également d'assurer l'unicité de présence de tous les processus lors de la lecture d'un changement de l'état du système. Le troisième et dernier sémaphore permet d'assurer la synchronisation au travers de l'application d'une opération Z, sur ce dernier, effectuant ainsi un point de rendez-vous entre tous les processus chargés de notifier.

3. Respect de la contrainte : *“Le client doit obtenir tout ce qu’il demande en même temps sinon il est mis en attente”*

Afin de respecter cette contrainte nous réalisons un *semop* sur un tableau de *sembuf*. Le tableau de *sembuf* correspond à la demande du client. Ce tableau ne contient pas nécessairement précisément les valeurs entrées par le client car comme vu sur le *point 1* les opérations de notre tableau de sembuf sont susceptibles d’être modifiées en fonction de l’état courant du système (c’est en fait uniquement le cas pour les demandes de ressources partagées). Dès lors, si toutes les demandes ne sont pas satisfaites en même temps, le client bloque jusqu’à satisfaction de sa demande.

4. Respect de la contrainte : *“Un client doit pouvoir demander une réservation de ressources impliquant plusieurs sites”*

Cette contrainte est liée à la précédente. En effet, lorsqu'un client saisit une demande de ressource sur un site donné, celle-ci ne doit pas être traitée immédiatement car sinon nous ne respectons plus la contrainte précédente. C’est en ce sens qu’une fois que le client affirme ne plus avoir de demande de réservation de ressource à réaliser, nous réalisons réellement sa demande qui peut donc concerner plusieurs sites.

5. Respect de la contrainte : *“L’état du système doit décrire pour chaque site et chaque type de ressource la quantitée réservée par utilisateur et le mode de réservation”*

Cette contrainte est étroitement liée à la conception de la structure descriptive du système. Afin de la respecter, nous avons inclus un tableau de client dans chacun des sites de notre système. Ainsi nous pouvons en permanence connaître le nombre de clients sur un site donné et les possessions en ressources exclusives et partagées de tous les clients du site.

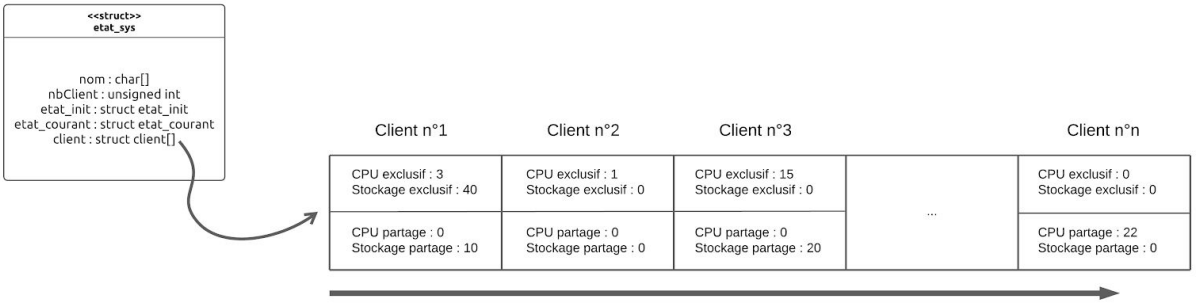


Figure 8 - Tableau de client présent dans chacun des sites de notre segment

Dès lors il ne nous reste plus qu'à parcourir le tableau pour obtenir toutes les infos de tous les clients.

6. Respect de la contrainte : *“Si un client est bloqué en attente de satisfaction de sa demande, un autre client doit pouvoir réserver des ressources”*

Cette contrainte est intrinsèque aux mécanismes d'exclusion des processus. Elle est liée à la manière dont les demandes des clients sont gérées. Pour notre système, cette contrainte est satisfaite. Dès lors qu’un client effectue une demande trop importante par rapport à l’état actuel du système, celui-ci est mis en attente. Si un autre client effectue la même demande mais bien moins importante et cohérente par rapport aux ressources disponibles dans le système, alors celui-ci est satisfait avant le client précédent.

7. Respect de la contrainte : *“Une mise à jour de l’état du système implique des envois parallèles de cette mise à jour aux différents clients”*

Pour respecter au mieux cette contrainte nous avons fait en sorte que toute demande ayant été satisfaite et entraînant une modification du segment est dupliquée sous la forme d’une chaîne de caractère dont la valeur est affectée au deuxième segment de mémoire partagé. En procédant ainsi nous pouvons ainsi extraire uniquement l’information intéressante (la demande ou la libération de ressource) et non pas un affichage intégrale du système. La notification du changement est ensuite réalisé par un *Thread* propre à chaque client

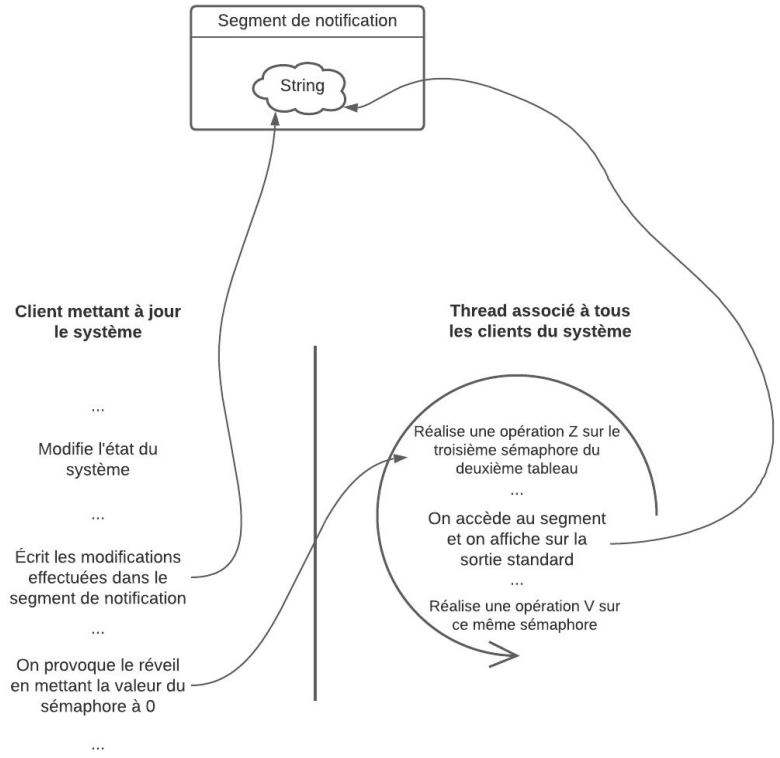


Figure 9 - Fonctionnement de la notification afin d’informer tous les clients

Comme nous pouvons le constater sur la *Figure 9*, le processus effectuant la modification du système provoque le réveil des Threads bloqué par une opération Z. Ce réveil est déclenché en mettant à 0 la valeur du sémaphore de notification. Les Threads quant à eux sont dans une boucle infinie. La première instruction qu'ils exécutent est cette opération Z. La valeur de ce sémaphore est initialisé par le serveur à 1. De fait, les threads bloquent tous immédiatement sur l'opération Z. Lorsqu'un processus met à jour le segment des sites, il écrit une copie des modifications dans le segment de notification et met ensuite à 0 la valeur du sémaphore de notification. Ceci provoque le réveil immédiat de tous les Threads en attente. Ils accèdent ensuite au segment de notification, affiche le contenu du segment sur la sortie standard (ce qui provoque la notification pour tous les clients) et réalise ensuite une opération V sur ce même sémaphore afin de s'assurer d'être de nouveau bloqué sur l'opération Z lors de la prochaine itération.

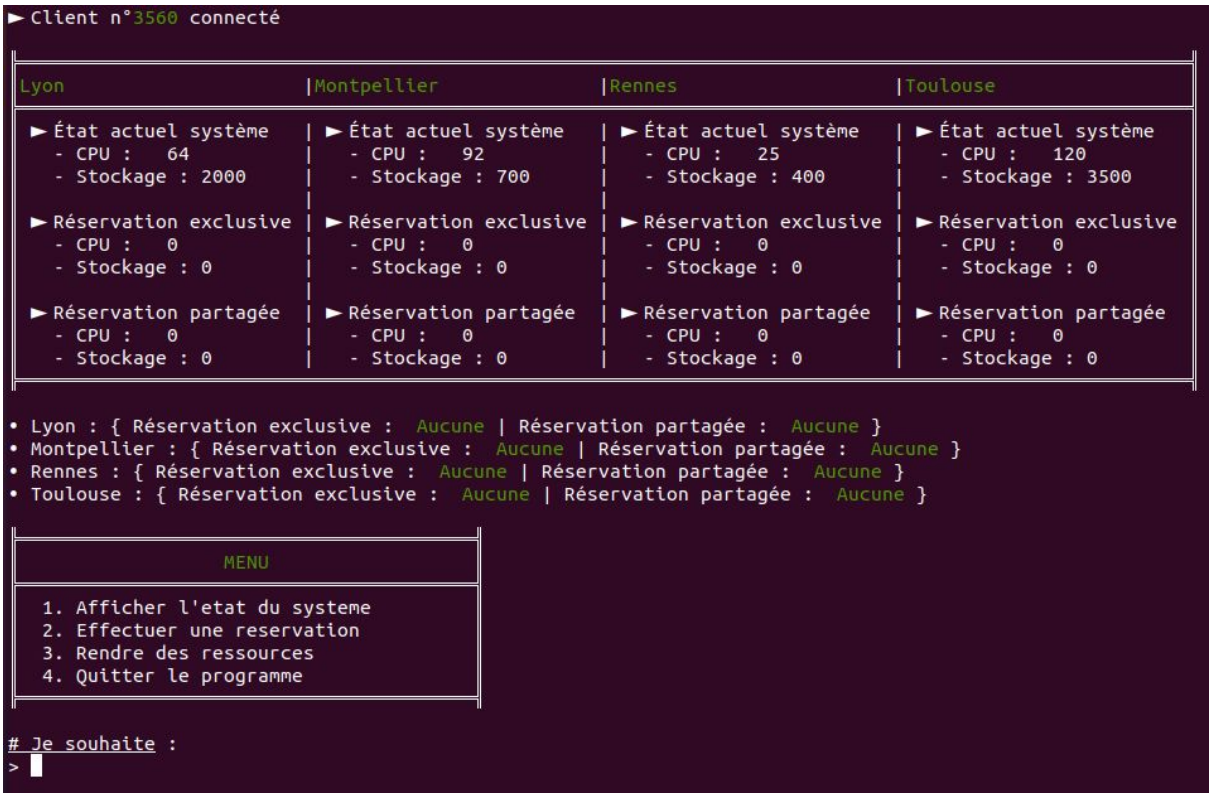
8. Respect de la contrainte : *"Gestion des erreurs en cas de suppression d'un objet IPC"*

Comme nous avons pu le voir en cours, la suppression d'un objet IPC en cours d'utilisation provoque un arrêt impromptu du processus en cours d'exécution et manipulant cet objet IPC (à l'exception des segments de mémoire partagée ou la suppression a lieu une fois que le dernier détachement s'est réalisé). Le serveur étant l'instance créatrice et destructrice des différents objets IPC, si celui-ci s'arrête inopinément alors tous les clients s'arrêteront au même moment. En provoquant volontairement un arrêt du serveur nous avons pu constater que c'est lors de l'appel à *semop* que les clients s'interrompent. Nous avons donc regardé le manuel de la fonction *semop* et nous avons vu que la fonction *semop* en cas d'échec renvoi -1 (ça nous l'avions vu en cours) mais précise également le type d'erreur. En l'occurrence ici, il s'agit d'une erreur de type *EIDRM* signifiant que le tableau de sémaphore a été supprimé. De fait, dans le corps du traitement d'erreur de la fonction *semop* nous avons rajouté une condition *if(errno == EIDRM)* ou *errno* est la variable statique contenant le type d'erreur. C'est ainsi que nous mentionnons au client que son arrêt est dû à un arrêt du fonctionnement du serveur. Le traitement des erreurs a aussi été fait sur les autres objets IPC mais en restant plus général sur le type d'erreur.

Spécificités et fonctionnalités de notre programme :

1. Fonctionnement général de notre système de réservation

Lorsqu'un client lance le programme il obtient l'affichage suivant :



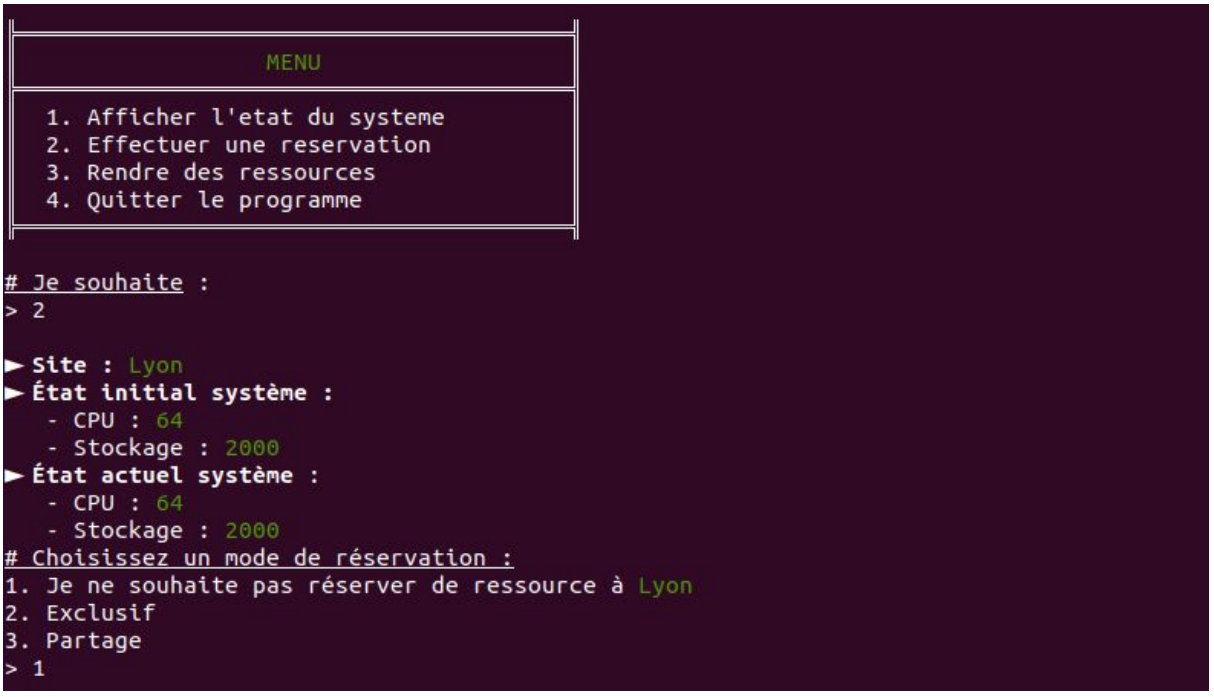
On peut constater 3 choses. Premièrement le client possède un identifiant (en haut à gauche), il s'agit du *PID* du processus ce qui nous assure l'unicité des identifiants des différents clients. Deuxièmement, tous les sites du système sont affichés à l'écran sous la forme d'un tableau et indiquent leur état actuel (quantité de ressources libres), les réservations exclusives (quantité de ressources exclusives utilisées) et les ressources partagées (quantité de ressources partagées utilisées). Troisièmement une liste à puce est présente sous le tableau et permet de savoir pour chaque site : quels clients ont loué des ressources, combien de ressources ont ils loué et avec quel mode de réservation.

Le nombre de sites est défini statiquement, autrement dit pour ajouter un site il faut l'ajouter manuellement dans la fonction d'initialisation des différents sites du système. En revanche, le nombre de clients par site peut être défini dynamiquement, nous l'avons fixé par défaut à 100 via une constante de préprocesseur.

En dessous de cette liste à puce nous avons le menu de navigation permettant au client d'effectuer différentes actions.

La première action permet de rafraîchir l'affichage du système afin de constater les différents changements qui ont pu avoir lieu.

La deuxième action permet de réaliser une réservation. Lorsqu'on réalise cette action, un listing des différents sites de notre système apparaît :



Dès lors, l'utilisateur est informé de l'état initial du site (pour éviter qu'il soit tenté de demander davantage de ressource qu'il n'y en a) et de l'état actuel (courant) indiquant les ressources disponibles. Bien évidemment nous avons pris en considération tous les cas types d'erreur à savoir :

- Le client demande plus de ressource qu'il n'y en a
- Le client rentre une valeur négative
- Le client rentre un type différent qu'une valeur entière
- Le client demande des ressources et peut s'auto bloquer

Le client a à présent 3 possibilités : soit il décide de ne prendre aucune ressource sur le site actuellement présent (dans ce cas on lui propose de prendre des ressources sur les autres sites). Soit il décide de prendre des ressources en mode exclusif, soit en mode partagé.

Supposons qu'il ne souhaite prendre aucune ressource sur le site de Lyon. On entre donc la valeur 1.

Il obtient ensuite une proposition de demande de ressource sur le site de Montpellier.
Supposons qu'il souhaite prendre 10 CPU et 300 Go de stockage, le tout en mode exclusif.

```
► Site : Montpellier
► État initial système :
   - CPU : 92
   - Stockage : 700
► État actuel système :
   - CPU : 92
   - Stockage : 700
# Choisissez un mode de réservation :
1. Je ne souhaite pas réserver de ressource à Montpellier
2. Exclusif
3. Partage
> 2

# Choisissez le nombre de CPU à réserver (/!\ MAX = 92) :
> 10

# Choisissez le nombre de Go de stockage à réserver (/!\ MAX = 700 Go) :
> 300
```

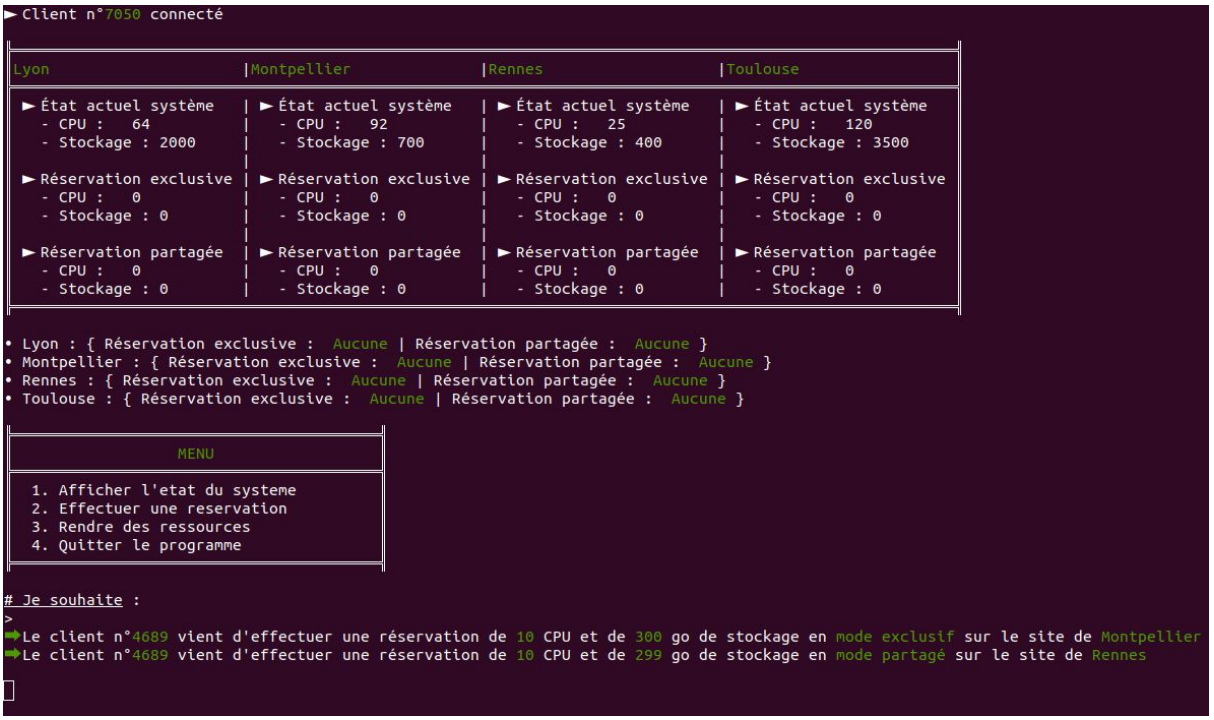
Sa demande ayant été saisi, on obtient une proposition pour le site suivant :

```
► Site : Rennes
► État initial système :
   - CPU : 25
   - Stockage : 400
► État actuel système :
   - CPU : 25
   - Stockage : 400
# Choisissez un mode de réservation :
1. Je ne souhaite pas réserver de ressource à Rennes
2. Exclusif
3. Partage
> 3

# Choisissez le nombre de CPU à réserver (/!\ MAX = 25) :
> 10

# Choisissez le nombre de Go de stockage à réserver (/!\ MAX = 400 Go) :
> 299
```

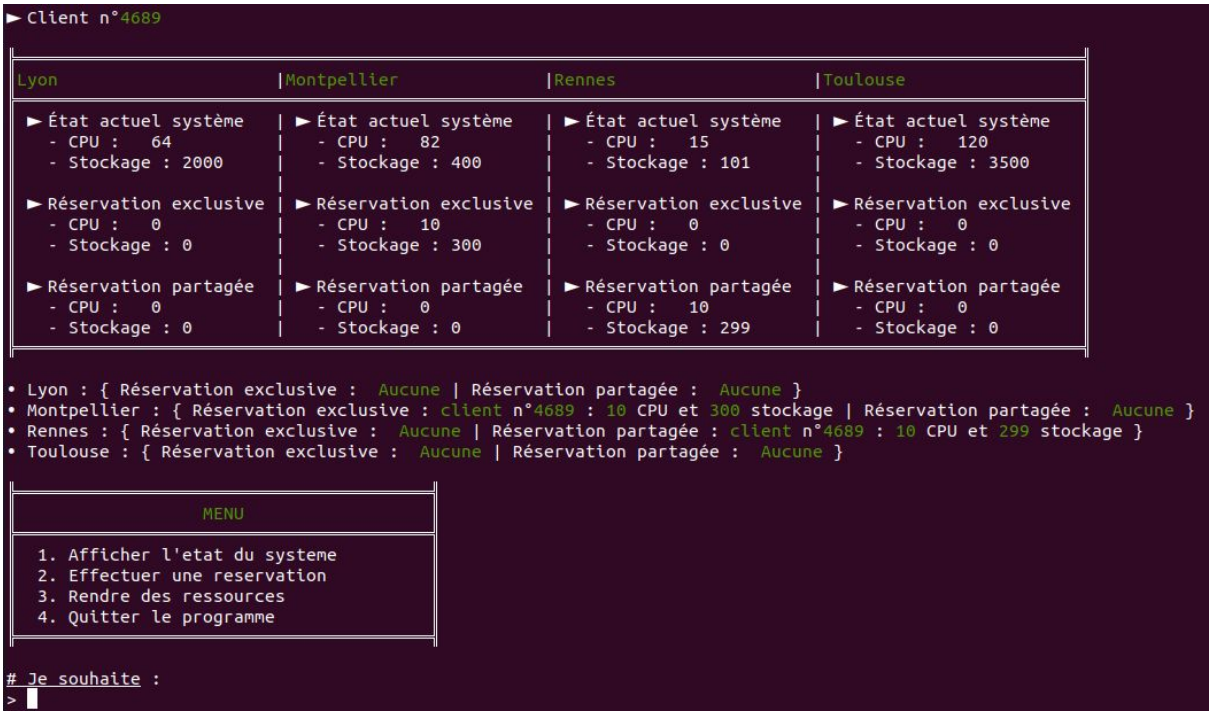
Cette fois-ci il décide de prendre 10 CPU et 299 Go de stockage à Rennes, le tout en mode partagé.
Il obtient une proposition de demande de ressource pour le site suivant (Toulouse) et décide de ne rien réserver.

```
► Site : Toulouse
► État initial système :
   - CPU : 120
   - Stockage : 3500
► État actuel système :
   - CPU : 120
   - Stockage : 3500
# Choisissez un mode de réservation :
1. Je ne souhaite pas réserver de ressource à Toulouse
2. Exclusif
3. Partage
> 1
```

Au même moment, un autre client présent sur le menu de navigation est informé de la demande qui vient d'être effectuée :



Le client ayant effectué la demande et lui aussi informé de sa propre demande.
Par ailleurs lorsque ce client affiche l'état du système dans le menu de navigation, il obtient le rendu visuel suivant :



On peut observer que l'état du système a été mis à jour. Pour rappel le client n°4689 avait demandé 10 CPU et 300 Go de stockage à Montpellier le tout en exclusif et 10 CPU et 299 Go de stockage à Rennes le tout en partagé.
De plus, on constate que dans liste à puce, l'élément Montpellier et l'élément Rennes ont aussi été mis à jour. Tous deux contiennent les informations du client possédant des ressources chez eux.

Désormais imaginons que le client n°7050 demande davantage de ressources que disponible sur les mêmes sites ou le client n°4689 a réservé (on prend cet exemple pour volontairement provoquer un blocage).

```
► Site : Montpellier
► État initial système :
   - CPU : 92
   - Stockage : 700
► État actuel système :
   - CPU : 82
   - Stockage : 400
# Choisissez un mode de réservation :
1. Je ne souhaite pas réserver de ressource à Montpellier
2. Exclusif
3. Partage
> 3

# Choisissez le nombre de CPU à réserver (/!\ MAX = 92) :
> 83

# Choisissez le nombre de Go de stockage à réserver (/!\ MAX = 700 Go) :
> 0


► Site : Rennes
► État initial système :
   - CPU : 25
   - Stockage : 400
► État actuel système :
   - CPU : 15
   - Stockage : 101
# Choisissez un mode de réservation :
1. Je ne souhaite pas réserver de ressource à Rennes
2. Exclusif
3. Partage
> 3

# Choisissez le nombre de CPU à réserver (/!\ MAX = 25) :
> 12

# Choisissez le nombre de Go de stockage à réserver (/!\ MAX = 400 Go) :
> 350
```

```
► Site : Toulouse
► État initial système :
   - CPU : 120
   - Stockage : 3500
► État actuel système :
   - CPU : 120
   - Stockage : 3500
# Choisissez un mode de réservation :
1. Je ne souhaite pas réserver de ressource à Toulouse
2. Exclusif
3. Partage
> 1
```

Celui-ci se retrouve bloqué. En effet, l'état du système est actuellement le suivant :

Lyon	Montpellier	Rennes	Toulouse
► État actuel système - CPU : 64 - Stockage : 2000	► État actuel système - CPU : 82 - Stockage : 400	► État actuel système - CPU : 15 - Stockage : 101	► État actuel système - CPU : 120 - Stockage : 3500
► Réservation exclusive - CPU : 0 - Stockage : 0	► Réservation exclusive - CPU : 10 - Stockage : 300	► Réservation exclusive - CPU : 0 - Stockage : 0	► Réservation exclusive - CPU : 0 - Stockage : 0
► Réservation partagée - CPU : 0 - Stockage : 0	► Réservation partagée - CPU : 0 - Stockage : 0	► Réservation partagée - CPU : 10 - Stockage : 299	► Réservation partagée - CPU : 0 - Stockage : 0

Et la demande effectué par le client est la suivante :

1. 83 CPU en partagé à Montpellier
2. 12 CPU et 350 Go de stockage à Rennes

La première demande ne peut pas être satisfaite. En effet, Montpellier dispose de 82 CPU libres et de 10 CPU utilisés en exclusif. Dès lors la demande de 83 CPU en partagé n'est pas satisfiable pour le moment.
En revanche la deuxième demande peut être satisfaite. En effet, Rennes dispose de 15 CPU et 101 Go de stockage libres mais également de 10 CPU et 299 Go stockage utilisés en mode partagé. Ainsi la demande n°2 du client ne ferait que rajouter 2 CPU et 101 Go de stockage utilisés en mode partagé.
Néanmoins étant donné la contrainte étudiée au point 3, le client est bloqué car l'ensemble de sa demande ne peut être satisfaite.

Maintenant nous allons volontairement rendre 1 CPU utilisé en mode exclusif à Montpellier par le client n°4689 afin d'observer le réveil de l'autre client (car dès lors l'intégralité de sa demande sera satisfiable).

Pour se faire il nous suffit de rentrer la valeur 3 sur le menu de navigation.

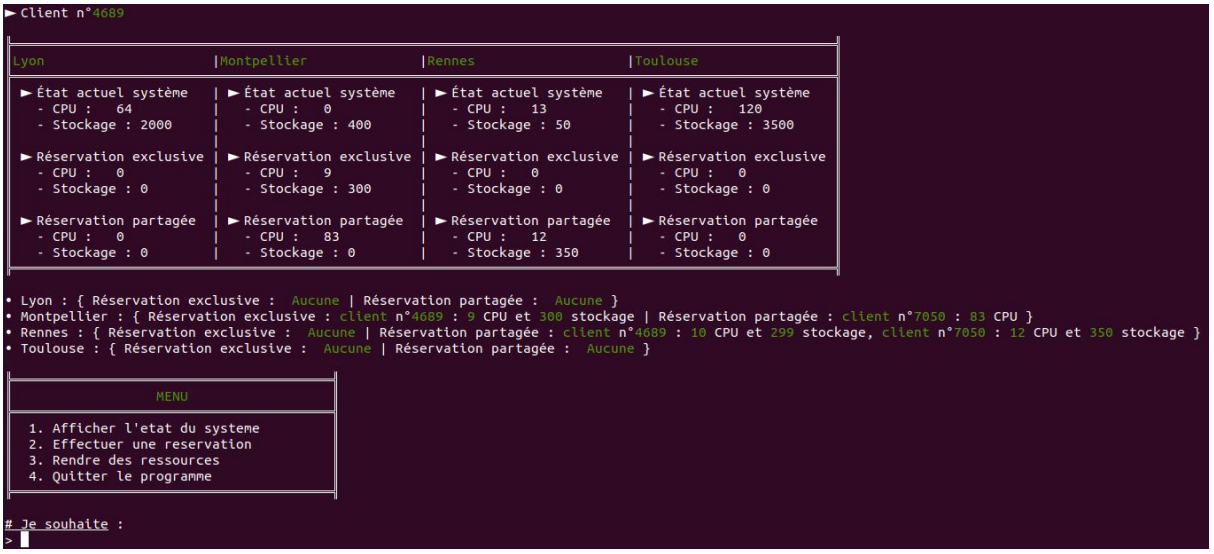
Dès lors, un nouveau listing des différents sites apparaît mais cette fois-ci pas de tous les sites, seulement des sites sur lesquels on possède des ressources.
De la même manière que pour la demande de ressource ou tout autre forme de saisie, la gestion des erreurs a été prise en compte. Par exemple ici, on peut constater un affichage nous indiquant que l'on dispose uniquement de 10 CPU et 300 Go de stockage en mode exclusif sur le site de Montpellier. De fait, si on demande à rendre des ressources partagées, un message d'erreur apparaît nous indiquant que nous n'en avons pas.

Nous rentrons donc la valeur 2 afin de rendre 1 CPU en mode exclusif sur le site de Montpellier. Nous décidons toutefois de ne pas rendre de ressources sur le site de Rennes.

Dès lors, nous sommes notifié de tous les changement suivants :

Nous sommes de notre propre libération de ressource. Par ailleurs nous sommes également notifié que le client qui était en attente vient d'obtenir toutes les ressources qu'il souhaitait.

Si nous affichons le nouvel état du système, nous obtenons :



Comme nous pouvons le constater, l'état du système a bien été mis à jour. Il en va de même pour la liste à puce ou on peut observer qu'un nouveau client a fait son apparition.

Lorsqu'un client décide de quitter le programme, celui-ci rend immédiatement toutes les ressources dont il possède :

Et les autres clients en sont immédiatement informés :

2. Principaux problèmes rencontrés

a) Le problème de l'auto blocage

Contexte :

Supposons qu'il n'y ait que le site de Lyon avec les ressources disponibles suivantes :

- 64 CPU
- 2000 Go de stockage

Un client C1 effectue une première demande en mode exclusif : 40 CPU et 0 Go de stockage.
Il effectue ensuite une deuxième demande toujours en mode exclusif : 25 CPU et 0 Go de stockage.
On constate un problème, ce dernier se bloque tout seul. Afin de pallier ceci, nous effectuons une vérification de la saisie. Si le client effectue une demande de ressource exclusive alors la vérification est :

Qt de ressource en exclusif (du client) + Qt de ressource en partagé du client + Qt demandé (en exclusif) > Qt de ressource initiale ?

Si l'assertion précédente est vraie alors une erreur est affichée, lors de la demande, sur la sortie standard.
Par exemple avec le cas précédent ça donnerait : 40+0+25 > 64 ce qui est vrai.
Maintenant si le client effectue une demande de ressource en partagé, la vérification est un tout petit peu différente :

Qt de ressource en exclusif (du client) + Qt demandé (en partagé) > Qt de ressource initiale ?

La raison de cette différence vient du fait que les ressources partagées ne sont pas additives du point de vue d'un client. Autrement dit, si un client demande 20 CPU en partagé puis demande 40 CPU de nouveau en partagé, il en aura pas 60 mais bien 40.

b) La déconnexion impromptue via l'utilisation de CTRL-C

Nous avons voulu prendre en compte le fait qu'un client et même serveur quitte via l'utilisation de CTRL-C. Le problème est que si on quitte le serveur de cette manière là, les objets IPC ne sont pas détruits. De la même manière, si on quitte un client de cette manière alors celui-ci ne rend pas les ressources qu'il possède et l'état du système devient donc incohérent. Afin de gérer ce problème nous avons inclu dans notre programme la bibliothèque *signal.h* permettant de gérer les signaux. Nous avons pris connaissance du fait que le signal envoyé lors d'un CTRL-C est un *SIGINT* qui est un type de signal signifiant une interruption logicielle. Dès lors, il nous suffit d'utiliser la fonction *signal* en lui passant en paramètre le type de signal, ici un *SIGINT* et une fonction (c'est elle qui sera appelée lors de la capture du CTRL-C). Dans cette fonction qui sera appelée, nous avons passé en

paramètre une structure qui contient pour le serveur, les id de tous les objets IPC à détruire et qui contient pour le client, un appel à la fonction de libération de ressource.

Exemple : extinction du serveur par un CTRL-C

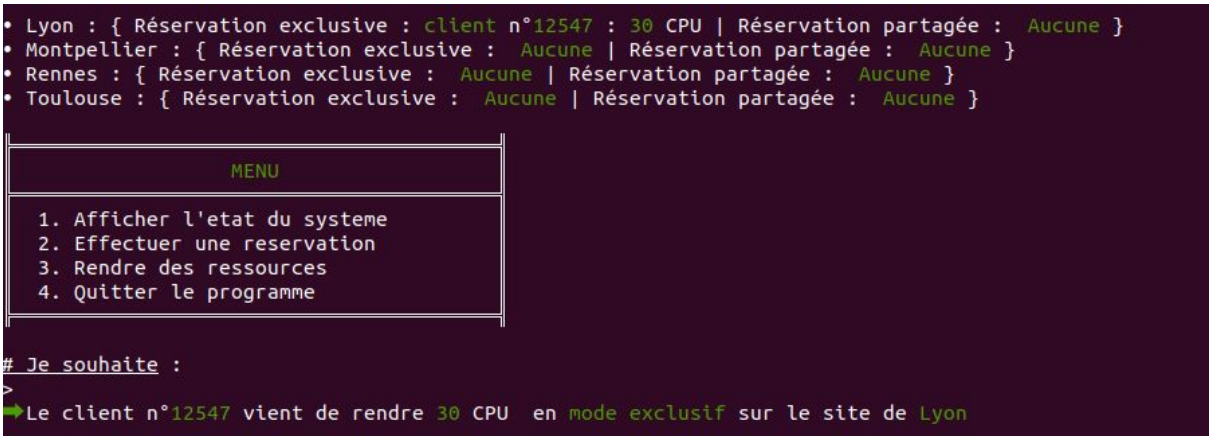
```
Serveur id : 12049
create shm ok [id : 294918]
create shm ok [id : 294931]
create sem ok [id : 6]
create sem ok [id : 7]
sem[0] = 64 (CPU)
sem[1] = 92 (CPU)
sem[2] = 25 (CPU)
sem[3] = 120 (CPU)
sem[4] = 2000 (Stockage)
sem[5] = 700 (Stockage)
sem[6] = 400 (Stockage)
sem[7] = 3500 (Stockage)
sem = 1 (verrou shm) [id : 7]
sem = 1 (verrou shm) [id : 7]
sem = 1 (verrou shm) [id : 7]
^C

Extinction prématurée du serveur...
Destruction des objets IPC en cours...

suppr shm ok [id : 294918]
suppr shm ok [id : 294931]
suppr sem ok [id : 6]
suppr sem ok [id : 7]

fin serveur
```

Le client est donc informé de la raison de sa déconnexion automatique :

```
• Lyon : { Réservation exclusive :  Aucune | Réservation partagée :  Aucune }
• Montpellier : { Réservation exclusive :  Aucune | Réservation partagée :  Aucune }
• Rennes : { Réservation exclusive :  Aucune | Réservation partagée :  Aucune }
• Toulouse : { Réservation exclusive :  Aucune | Réservation partagée :  Aucune }

╔═══════════════════════════════════════╗
║                 MENU                  ║
╠═══════════════════════════════════════╣
║  1. Afficher l'etat du systeme        ║
║  2. Effectuer une reservation         ║
║  3. Rendre des ressources             ║
║  4. Quitter le programme              ║
╚═══════════════════════════════════════╝

# Je souhaite :
>

Le serveur vient de se déconnecter, cela entraine une déconnexion du client.
```

De la même manière si on applique un CTRL-C sur un client :

Et on peut observer que les autres clients reçoivent bien la notification explicitant que le client qui vient de se déconnecter a préalablement rendu ses ressources :



Pour résumer les différentes fonctionnalités de notre programme :

- Le client peut effectuer une réservation en exclusif et/ou partagé sur un ou plusieurs sites en une seule fois
- Le client n'est pas obligé de rendre toutes ses ressources, il peut en rendre uniquement partiellement ou toutes s'il le souhaite
- Les erreurs de saisies, les erreurs liées aux objets IPC sont prises en compte
- Tous les clients sont notifiés d'un changement du segment
- Il est possible de consulter l'état du système et de visualiser quel(s) client(s) possèdent quoi et dans quel(s) site(s)
- Lorsqu'un client quitte, il rend ses ressources préalablement
- Lorsque le serveur quitte, les clients quittent proprement
- Les interruptions inopinées CTRL-C ont été géré

Bilan et conclusion :

Ce projet fut enrichissant et nous a permis de manipuler en profondeur les concepts que nous avions vu en cours et en TD. Il a éclairci certains aspects qui nous semblaient ambigu. Nous avons rencontré plusieurs difficultés et nous avons bien pris conscience du fait de modéliser, conceptualiser, visualiser l'architecture logicielle avant d'essayer de la mettre en place. Nous avons également bien entamé la version décentralisée toutefois celle-ci n'étant pas totalement fonctionnelle c'est avec quelques regrets que nous rendons la version centralisée. Néanmoins nous avons pris énormément de plaisir à la réaliser et nous comptons la continuer malgré tout.