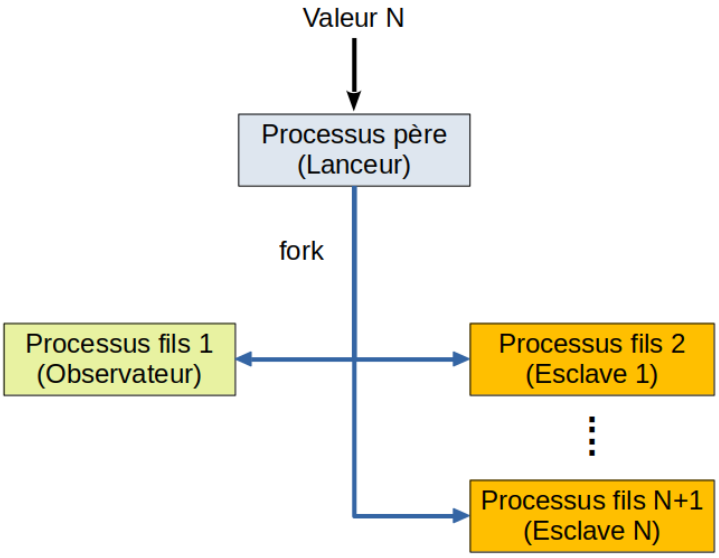
Mini-projet d'ICR 2023

Le mini-projet d'informatique concurrente et répartie sera réalisé par groupe de 3. Un espace sera créé sur Moodle pour déposer vos travaux. La date limite est fixée au mercredi 21 juin.

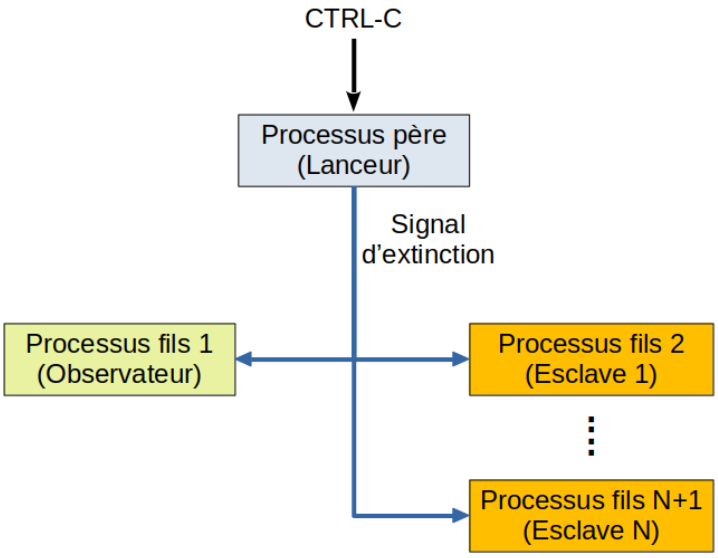
1. Architecture générale

Lors du lancement du programme, le processus « Père », qui joue le rôle de lanceur, va générer N+1 fils. N est un nombre fourni au programme de diverses manières (lors du lancement, en demandant à l'utilisateur, ou via un fichier de configuration) :

- le premier fils joue le rôle d'observateur ;
- les N autres fils jouent le rôle d'esclaves.



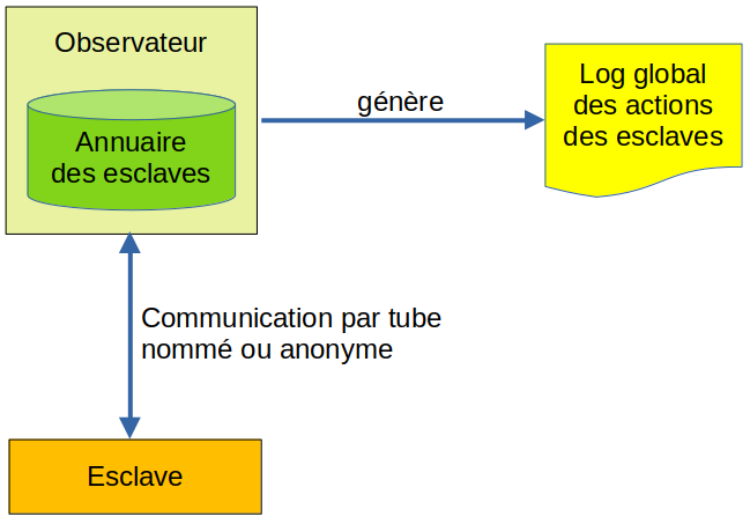
Le CRTL-C doit être géré par le père qui enverra un signal d'extinction vers ses fils afin que ces derniers s'arrêtent correctement. L'observateur doit être le dernier à s'arrêter. Une fois que tous les fils sont morts, le lanceur peut s'arrêter.



L’observateur

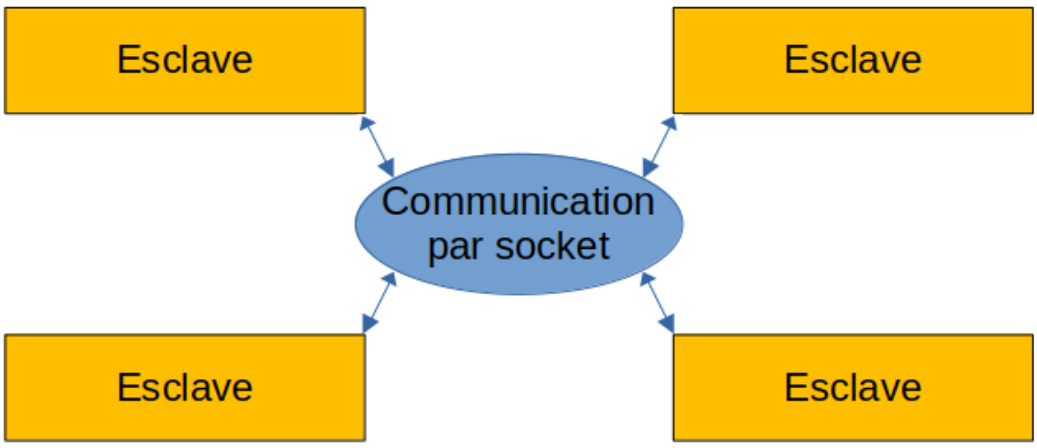
Ce processus est chargé de :

- gérer un annuaire des esclaves (une structure de données organisée selon vos besoins)
- gérer les traces (des actions) envoyées par les esclaves pour générer un fichier de log global à l’ensemble des esclaves



Les esclaves

Ces processus doivent communiquer entre eux par l’intermédiaire de sockets (locale ou TCP/IP)



Ce type de processus comporte en interne 4 threads POSIX :

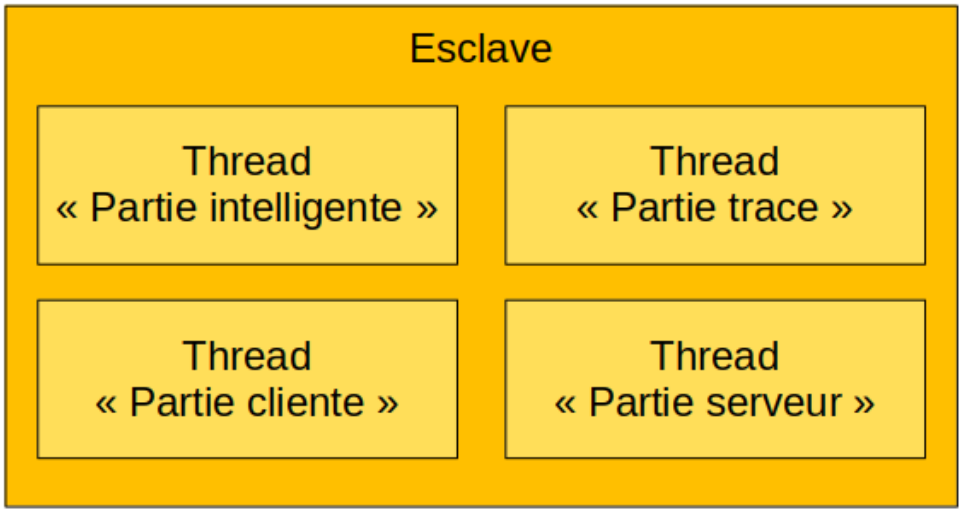
- un thread « partie intelligente »
- un thread « partie trace »
- un thread « partie cliente »
- un thread « partie serveur »

Le thread « partie intelligente » contient le code permettant de gérer le mécanisme d'exclusion mutuelle distribuée de Lamport.

Le thread « partie trace » va communiquer avec l'observateur via des tubes (anonymes ou nommés) pour :

- envoyer des traces des actions accomplies par l'esclave ;
- obtenir des informations sur le voisinage (connaître le nombre de voisins et comment communiquer avec eux via les sockets)

Les thread « partie client » et « partie serveur » permettent de gérer la communication entre les esclaves



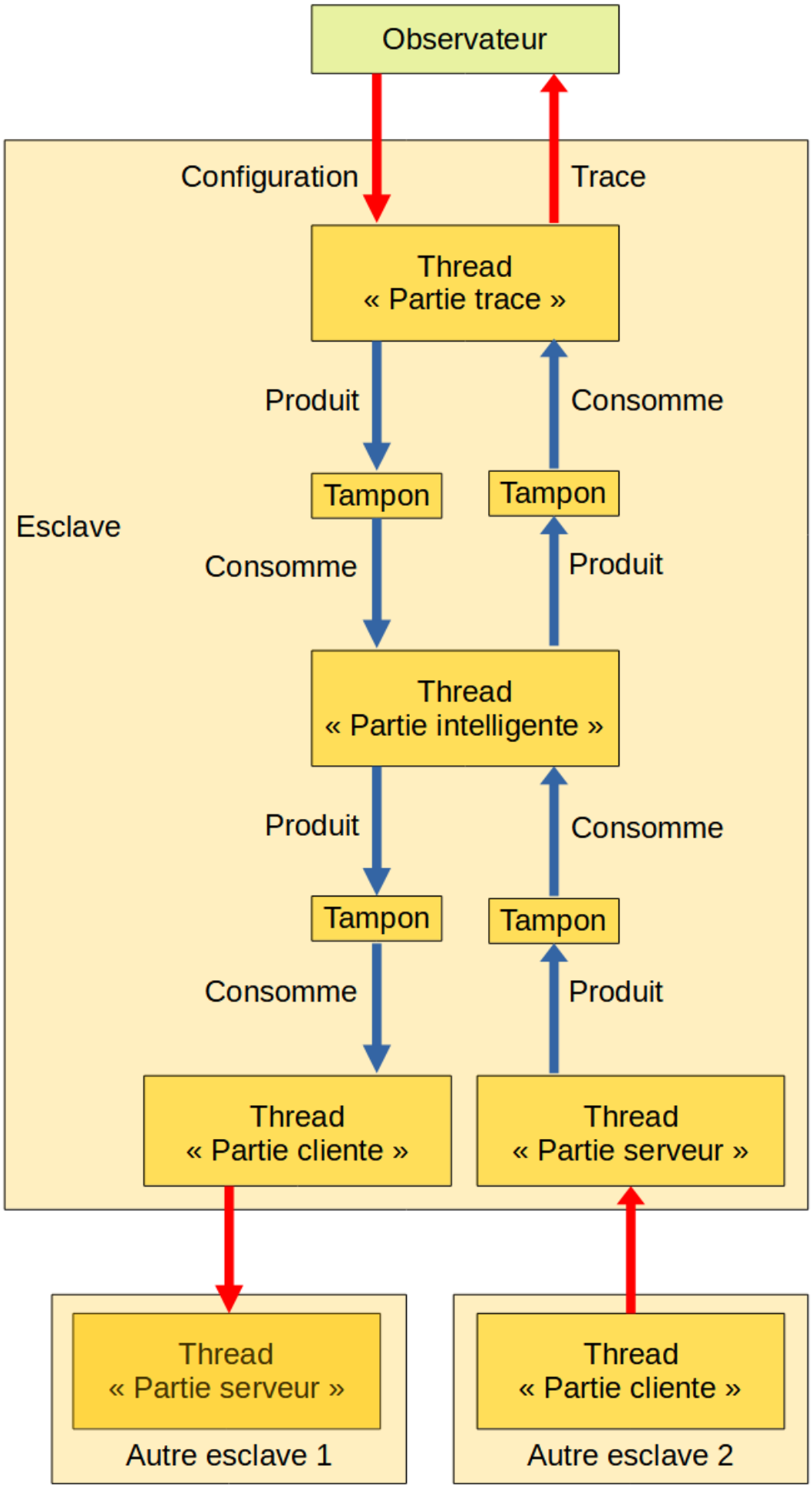
Ces différents threads communiquent entre eux en utilisant le mécanisme du producteur-consommateur (flèches bleues + tampons) :

- Le thread « partie intelligente » produit des infos de traces qui sont consommées par le thread « partie trace » afin de les envoyer vers l'observateur ;

- Le thread « partie intelligente » consomme des infos de configuration qui sont produites par le thread « partie trace » après les avoir reçues de l'observateur ;

- Le thread « partie intelligente » produit des données (envoyées vers des esclaves) qui sont consommées par le thread « partie client » afin de les envoyer vers le thread « partie serveur » de l'esclave cible ;

- Le thread « partie intelligente » consomme des données (reçues depuis des esclaves) qui sont produites par le thread « partie serveur » lorsqu'il reçoit des éléments des threads « partie client » des autres esclaves


Observateur
Configuration
Trace
Thread
« Partie trace »
Produit
Consomme
Tampon
Tampon
Esclave
Consomme
Produit
Thread
« Partie intelligente »
Produit
Consomme
Tampon
Tampon
Consomme
Produit
Thread
« Partie cliente »
Thread
« Partie serveur »
Thread
« Partie serveur »
Autre esclave 1
Thread
« Partie cliente »
Autre esclave 2

2. Travaux à réaliser

a. Mise en place de l'infrastructure décrite précédemment.

Cette infrastructure doit pouvoir se déployer quelque soit la valeur de N fournie.

Les traces « démarrage » et « arrêt » doivent être envoyées par les esclaves.

L'observateur doit gérer ces 2 traces de la manière suivante :

- La trace « démarrage » :
    - l'observateur ajoute des informations dans l'annuaire (par exemple, le PID de l'esclave, le nom du fichier d'échange pour la socket serveur locale) ;
    - l'observateur écrit dans le fichier de log que l'esclave a démarré ;
    - l'observateur retourne les informations de configuration (le nom du fichier d'échange pour la socket serveur locale).
- La trace « arrêt » :
    - l'observateur retire, de l'annuaire, les informations concernant l'esclave qui s'arrête ;
    - l'observateur écrit dans le fichier de log que l'esclave s'est arrêté ;
    - l'observateur retourne un acquittement vers l'esclave.

Les sockets serveur doivent être configurées et lancées au démarrage.

Le système doit pouvoir s'arrêter correctement lorsque le lanceur (le processus) père reçoit le CTRL-C. Le lanceur doit être le dernier fils à s'arrêter. Le père peut s'arrêter lorsque tous ses fils se sont arrêtés.

b. Test de la communication inter-esclaves

La partie intelligente doit (de façon continue) :

- demander à l'observateur des informations concernant le voisinage ;
- déterminer, au hasard, un esclave destinataire d'un message « PING » ;
- déposer, dans le tampon d'envoi, les informations permettant cet envoi ;
- déposer dans le tampon de trace, les informations relatives à cette action ;
- vérifier le contenu du tampon de réception ;
- récupérer le cas échéant l'information ;
- déposer dans le tampon de trace, les informations relatives à cette seconde action.

La partie client doit (de façon continue) :

- récupérer les informations d'envoi du tampon ;
- se connecter au bon serveur ;
- envoyer le message.

La partie serveur est en attente. Dès qu'elle reçoit une demande de connexion, elle récupère l'information et la dépose dans le tampon de réception, puis se remet en attente.

c. Implémentation du mécanisme d'exclusion mutuelle distribuée de Lamport

L'implémentation de ce mécanisme se fera principalement au niveau du thread « partie intelligente ». L'idée est de choisir au hasard des actions pour se rapprocher d'un comportement en conditions réelles. Le comportement de ce thread peut donc être décrit par le pseudo-code qui suit.

partieIntelligente ()

```
- Tant que ( ! fin )  faire

    * traiterMessagesReçus ()
    * faireAction ()
```

traiterMessagesReçus ()

```
- Boucle sur tous les messages présents dans le tampon de réception

    * lire un message dans le tampon de réception
    * incrémenter horloge

    * selon message

       - message simple :

           * envoyerTrace ( RECEPTION_MESSAGE , date , idEmetteur , message )

       - request        :

           * envoyerTrace ( RECEPTION_REQUEST , date , idEmetteur )
           * ajouter la demande dans la file d'attente, en triant d'abord par
             estampille, puis par id des émetteurs
           * incrémenter horloge
           * envoyer ( REPLY , date , idRecepteur )
             avec idRecepteur = idEmetteur qui a envoyé le request
           * envoyerTrace ( ENVOI_REPLY , date , idRecepteur )

       - reply          :

           * envoyerTrace ( RECEPTION_REPLY , date , idEmetteur )
           * incrémenter nbReply
           * tenterSectionCritique ()

       - release        :

           * envoyerTrace ( RECEPTION_RELEASE , date , idEmetteur )
           * tenterSectionCritique ()
```

tenterSectionCritique ()

```
- si nbReply OK et requête en premier dans la file d'attente

   * incrémenter horloge
   * envoyerTrace ( SECTION_CRITIQUE , date )
   * attendre une durée prise au hasard
   * incrémenter horloge
   * diffuser RELEASE
   * envoyerTrace ( DIFFUSION_RELEASE , date )
   * mode = "normal"
```

faireAction ()

```
  - si mode == "normal"

     * selon ( un nombre au hasard pour déterminer action à réaliser )

        - action locale :

            * incrémenter horloge
            * envoyerTrace ( ACTION_LOCALE , date )

        - envoi message :

            * incrémenter horloge
            * déterminer idRecepteur au hasard
            * envoyer ( message , date, idRecepteur )
            * envoyerTrace ( ENVOI_MESSAGE , date , idRecepteur, message )

        - request       :

            * mode = "demande de SC"
            * initialiser nbReply et file d'attente
            * diffuser REQUEST
            * envoyerTrace ( DIFUSSION_REQUEST , date )


Remarque : envoyerTrace () envoie l'ID du processus concerné par la trace même
           si ce n'est pas indiqué (c'est un argument implicite)
```

3. Contraintes de codage

Le travail doit être réalisé en C ou C++. En conséquence l'utilisation des classes est autorisée.

C++ propose ses propres implémentations des sémaphores… Il est possible de les utiliser à conditions d'utiliser au moins une fois les équivalents en C.

Exemple :

Il y a quatre liens « producteur-consommateur » à mettre en place, vous devez en implémenter au moins un en C, comme présenté durant le cours. Les autres peuvent être réalisés en C++.