INTER

Mode Client-Serveur

24 octobre 2012

Table des matières

Chapitre 1

INTER ET LA COMMUNICATION INTER–PROCESS

1.1 MODE SERVEUR

Inter peut travailler en mode serveur, c'est à dire que les commandes qu'il exécute ne lui sont plus fournie par le clavier, mais envoyées par des clients (programmes indépendants) communiquant avec lui par l'intermédiaire d'une mémoire partagée, que l'on appellera bloc de communication.

Un nombre indéfini de clients peuvent entrer en communication avec Inter et partager les matrices de travail. Les clients en attente sont mis en queue avant leurs traitements. Il n'y a pas de priorité, chacun est pris dans son ordre d'arrivée. La synchronisation est réalisée au moyens de sémaphores.

On dira qu'un client prend la main lorsque qu'il est autorisé à communiquer, et qu'il rend la main lorsque que son travail de client est terminé et qu'il laisse l'accès au serveur à d'autres clients.

Lorsqu'un client veut prendre la main, il attend que le serveur soit accessible. Durant cette phase d'attente, dont la durée peut être négligeable, le client est stoppé jusqu'au moment où le serveur devient libre.

La main est rendue selon deux schémas :

1) **En mode NOWAIT,** une fois que le client à la main, il envoie une commande au serveur. La commande est immédiatement exécutée et lorsqu'elle est terminée **c'est le serveur qui rend la main.** L'avantage de cette méthode est que le client peut continuer son propre processus sans se soucier de la durée d'exécution de la procédure. Par contre, il ne sais pas si la procédure c'est bien déroulée. C'est le prochain client (ou lui-même dans le cas d'un processus bouclant) qui saura s'il y a eu une erreur durant la précédente procédure.

Dans ce mode, il faut absolument utiliser des procédures si on doit envoyer plusieurs commandes consécutives. En effet, si on envoie des commandes une à une, on prend le risque de voir se glisser les commandes d'un autre client à un moment inaproprié.

2) **En mode WAIT,** le client indique au serveur qu'il ne doit pas rendre la main en fin de travail et que **c'est le client qui rendra la main.** Dans ce mode, le client se met en attente sur le serveur jusqu'au moment où ce dernier termine. Le client à la possibilité d'exécuter du travail localement avant de se mettre en attente. Ce mode permet d'enchaîner plusieurs exécutions de procédures et de tester la réussite de chaque procédures. Ce mode

permet aussi de passer la main de client en client (processus différents). Dans ce dernier cas, c'est le dernier client qui devra **obligatoirement** rendre la main. Le client à la possibilité d'exécuter du travail localement avant d'attendre la fin de la procédure.

Pour travailler dans le mode serveur, on lance Inter de la manière suivante :

```
inter -server [-block <bloc_de_données>] [-echo]
```

sachant que
– Le bloc "block.ref" est lu par défaut si aucun bloc n'est donné.
– L'option "-echo" permet l'écho des commandes arrivant à Inter.
– Les clients et Inter peuvent être lancés dans n'importe quel ordre, les client resteront en attente jusqu'a que le Inter soit prêt à accepter les commandes.

Inter peut savoir s'il a été lancé en mode serveur ou client :

SHMSRV()
Retourne 1 si Inter a été lancé en mode serveur.

SHMCLI()
Retourne 1 si Inter a été lancé en mode client.

Un client envoie donc des commandes et peut envoyer ou recevoir des paramètres. Ces paramètres sont accessibles par un mot-clé et sont écrits dans une zone de mémoire partagée appelée ici "bloc de communication". Un mot-clé est réservé, c'est :

COMMAND doit contenir la commande que le serveur exécutera

1.1.1 Accès au bloc de communication

Inter accède les mot-clés et leurs contenus avec les fonctions suivantes :

SHMINIT()
initialise le bloc de communication en le vidant.

SHMGET(key)
lit un paramètre référencié par *key* dans le bloc de communication.nx=ator(shmget("XSIZE"))

SHMPUT(key,content)
écrit un paramètre référencié par *key* et son contenu *content* dans le bloc de communication et initialise le bloc de communication.Après cette opération, le bloc ne contient qu'un paramètre.
Exemple : `dummy=SHMPUT("XSIZE",itoa(nx))`

SHMADD(key,content)
ajoute un paramètre référencié par *key* et son contenu *content*, dans le bloc de communication .

SHMSHOW()
visualise le bloc de communication à l'écran.

Attention, avant chaque appel à un serveur, il faut impérativement commencer par initialiser le bloc de communication, soit avec SHMINIT(), soit avec SHMPUT(). En effet, le bloc se remplis de façon incrémentale, il n'y a pas de mode *remplacement*.

On remarque, dans les exemples ci-dessus, que toutes les variables sont passées sous forme de chaîne de caractère, les expressions numériques sont à formatter, soit avec la fonction **itoa()** soit avec **format()**, ces expressions peuvent être lues avec la fonction **ator()**.
– Attention il faut respecter les minuscules et majuscules pour nommer le mot-clé.
– Si pour une raison ou une autre on désire utiliser uniquement **shmadd()**, il faut initialiser le bloc avec la fonction **shminit()**.
– A part **shmget()** qui retourne toujours une chaîne de caractère, les autres fonctions retournent toujours la valeur 0.

Par exemple, un client qui veut connaître les tailles des matrices accessibles lancera une procédure contenant les commandes suivantes qui une fois exécutée auront remplis le bloc de communication. Le client pourra lire le résultat dès que Inter finis la procédure. Exemple :

```
set tmp=shmini()
do i=1,nbmat()
   set shmadd(lcat("NX",itoa(i)),nx(i))
   set shmadd(lcat("NY",itoa(i)),ny(i))
enddo
```

1.1.2 Partage des matrices

Une partie des matrices peut être mis en mémoire partagée. Pour réaliser cela, il faut poser la variable SHAMEM à 1, donner la taille désirée aux matrices destinés à la mémoire partagée dans les variables MATSIZ(i) et préciser dans MATCOU(i) le nombre de couches utilisées par chaque matrice destinée à être placée en mémoire partagée.
Ces variables doivent être définies avant d'utiliser Inter en mode server. **Il est interdit de changer la taille de matrice partagée en cours de travail.**

1.2 MODE CLIENT

Ce mode permet de se connecter sur un autre Inter travaillant en mode serveur. Cela permet de faire un client Inter à part entière ou de fabriquer un simulateur de client.

Pour travailler dans ce mode, on lance Inter de la manière suivante :

```
inter -client
```

Dans le cas ou les sémaphores ou le bloc de communication sont détruits, le client peut se reconnecter au serveur avec la fonction :

CONNECT()
Se (re)connecte sur les sémaphores du serveur courant s'ils ont été tués.

Comme tout client, cet Inter récupère la mémoire partagée comprenant les matrices de travail et le bloc de communication. Comme c'est un client, c'est à l'utilisateur de gérer tout les problèmes de synchronisation pour la communication entre ces deux processus.
Pour réaliser cela on a à disposition une série de fonctions :

D'abord, deux fonctions d'usage simple, permettant de passer des commandes au serveur dans les cas les plus courants. Attention, ces fonctions ne permettent pas de récupérer des paramètres, car le client n'a plus la main au retour de ces fonctions et donc le bloc de communication peut déjà être corrompu par un autre client. Ces fonctions sont toutefois très utiles en mode interactif. Ce sont :

SHMCMD(<cmd>[,<timeout>])
Suspend le client, envoie une commande et libère le client.Retourne un status différent de zéro si il y a eu une erreur sur la commande **précédente**. Dans ce cas (1=erreur sur serveur, 2=serveur déconnecté, 3=<CTRL>–C sur serveur, 101=SIGHUP, 102=SIGINT (<CTRL>–C), 113=SIGPIPE, 114=SIGALRM (timeout)) aucun message n'est affiché et l'erreur interne n'est pas activée
Exemple : `i=shmcmd("@qq")`

SHMCMDW(<cmd>[,<timeout_wait>[,<timeout_cmd>]])
Suspend le client, envoie une commande et attend la fin de la commande pour libérer le client.Retourne un status différent de zéro si il y a eu une erreur sur la commande en cours. En cas d'erreur (1=erreur sur serveur, 2=serveur déconnecté, 3=<CTRL>–C sur serveur, 101=SIGHUP, 102=SIGINT (<CTRL>–C), 113=SIGPIPE, 114=SIGALRM (timeout)) une message est affiché mais l'erreur interne n'est pas activée

Les autres fonctions sont des fonctions de bases qui permettent de construire tout les type de dialogues possible. Ce sont :

SHMWAIT([<timeout>])
suspend le client jusqu'à que le serveur soit prêt.(décrémente le sémaphore #0). Bloque le client si le serveur n'est pas prêt. Retourne un status différent de zéro si il y a eu une erreur sur la commande **précédente**. Dans ce cas (1=erreur sur serveur, 2=serveur déconnecté, 3=<CTRL>–C sur serveur, 101=SIGHUP, 102=SIGINT (<CTRL>–C), 113=SIGPIPE, 114=SIGALRM (timeout)) aucun message n'est affiché et l'erreur interne n'est pas activée

SHMCONT()
indique au serveur d'exécuter la commande placée dans "COMMAND" et lui signale de rendre la main.(incrémente le sémaphore #1)Le serveur se libérera par lui-même à la fin de la commande.

SHMACK()
indique au serveur d'exécuter la commande placée dans "COMMAND" et lui signale de ne pas rendre la main.(pose le sémaphore #2 à 1 et incrémente le sémaphore #1).

SHMWACK([<timeout>])
attend que le serveur ait finis (après un SHMACK()).(Attend que le sémaphore #2 soit égal à zéro). Suspend le client tant que sa tâche n'est pas terminée. Retourne un status différent de zéro si il y a eu une erreur sur la commande en cours. En cas d'erreur (1=erreur sur serveur, 2=serveur déconnecté, 3=<CTRL>–C sur serveur, 101=SIGHUP, 102=SIGINT (<CTRL>–C), 113=SIGPIPE, 114=SIGALRM (timeout)) une message est affiché mais l'erreur interne n'est pas activée

SHMFREE()
Rend la main (après un SHMWACK()).(incrémente le sémaphore #0) Apres cette commande, le serveur est accessible pour un autre client

L'utilisation des ces commandes se fait selon 2 principaux schémas :
1) Le premier ou l'on envoie une commande sans se soucier du résultat :

```
local dummy
! attend que le serveur soit prêt
dummy=shmwait()
! initialise le bloc de communication
dummy=shmini()
! y place une commande
dummy=shmadd("COMMAND","@proc")
! indique au serveur d'exécuter la commande
dummy=shmcont()
......
```

1) Le second ou l'on envoie une commande et on attend une réponse :

```
local dummy status nval
! attend que le serveur soit prêt
dummy=shmwait()
! initialise le bloc de communication
dummy=shmini()
! y place une commande
dummy=shmadd("COMMAND","@proc")
! indique au serveur d'exécuter la commande mais en
! attendant une réponse
dummy=shmack()
! en attendant on peut exécuter localement autre chose
......
! on attend une réponse et on teste sa réussite
status=shmwack()
if (status) goto erreur
! on lit un message en retour (par exemple)
nval=ator(shmget("NVAL"))
! on libere le serveur pour un autre client
dummy=shmfree()
......
```

Il faut bien prêter garde que si l'on passe outre le système de synchronisation, les valeurs que l'on va lire ou écrire dans le bloc de communication ou dans les matrices peuvent interférer avec d'autre clients utilisant eux le système de synchronisation. Ce mode de fonctionnement est à utiliser avec prudence.

De plus la taille de la mémoire partagée dépend des matrices définies en mémoire partagée de l'inter travaillant en mode serveur. Donc lors d'une connection avec un Inter client, utilisant un bloc de données contenant les paramètres de matrices différents, il faut récupérer (par procédure) les valeurs des variables du serveur. Sauver ces valeurs, sortir du client et le relancer pour récupérer les bonnes matrices. Pour réaliser cela on utilise (cote client) la procédure "getmat.prc"

1.3 COMPORTEMENT EN CAS DE PROBLÈME

Voici la liste des propriétés et comportements des clients et serveurs dans des situations extraordinaires :

– Un serveur à la propriété de pouvoir survivre à la perte de ses sémaphores (après un "ipckill" par exemple), il arrive à les refabriquer.
– Si les sémaphores sont tués deux fois de suite sans qu'aucune communication n'a eu lieu, le serveur se "suicide".
– Si un serveur est tué sans qu'il soit en communication, aucun problème.
– Si un client essaie de se connecter sur un serveur inexistant, le client est suspendu et sa commande sera

exécutée lorsque le client sera lancé.
– Si un serveur est tué pendant une communication avec un client en attente. Le client reste en attente jusqu'au moment ou le serveur est relancé.
– Si un serveur est tué pendant une communication sans client en attente. Aucun problème.
– Un client à la propriété de pouvoir se connecter à un serveur à n'importe quel moment. Il peut se déconnecter que s'il n'est pas en communication avec un serveur. Les clients peuvent ainsi être lancés et tués un nombre de fois illimité
– Si un client en attente est tué, le sémaphore ne sera pas libérer. Il faut donc le libérer avec un shmfree() sur un autre au client (voir aussi l'utilitaire IPCSTAT).

1.4 UTILISATION DE PLUSIEURS SERVEURS

Un client peut converser avec plusieurs serveurs, les serveurs de ce client peuvent être simultanément clients d'autres serveurs et de serveurs du client principal. La complexité du système est limitée que par l'imagination de l'utilisateur.
Pour mettre en oeuvre des systèmes clients–serveurs entrelacés, il faut se souvenir qu'un bloc de communication ainsi que les sémaphores qui y sont associés appartiennent au serveur. Ainsi chaque serveur a son propre bloc de communication et ses propres sémaphores. La manière de différencier plusieurs blocs de communication et sémaphores est donnée par l'attribution d'une clé (valeur numérique) différentes à chaque serveur. La syntaxe d'appel à Inter en mode serveur devient :

```
inter -server [<clé>] [autres options]
```

Attention, la clé est un entier positif arrondi à la centaine. La clé par défaut est 1000.

Le client doit donner un nom symbolique aux serveurs avec qui il veut converser ainsi que le clé du serveur. La syntaxe d'appel à Inter en mode client devient :

```
inter -C<nom_du_serveur_1> <clé_1>  [...] -C<nom_du_serveur_n>
                                          <clé_n> [autres options]
```

Attention, il ne faut pas mettre d'espace entre le `-C` et le nom du serveur.

Par exemple, si la configuration est la suivante :
– un serveur du système téléscope (nom :tele)
– un serveur du système guidage (nom :guid), ce système est aussi client de tele.
– un serveur du spectrographe (nom :elodie)

– un client principal ayant comme serveur : tele, guid et elodie

Les appels de lancements seront :

```
inter -server 1200
inter -server 1300 -Ctele 1200
inter -server 1400
inter -Ctele 1200 -Cguid 1300 -Celodie 1400
```

Les fonctions de communication restent les mêmes que pour une communication client–serveur unique. Seul un appel doit être rajouté pour permettre de sélectionner le bon serveur. Cet appel est :

SELECT(<server>)
Sélectionne le serveur sur lequel vont travailler toutes les fonctions de communication.
Exemple : `i=select("ccd")`

On peut également visualiser l'ensemble des serveurs à disposition avec :

SHOWSEL()
Affiche le nom de tous les serveurs possible ainsi que le serveur actuellement sélectionné.
Exemple : `i=showsel()`

Par exemple, pour envoyer la commande "@test" aux serveurs "tele" et "elodie", la procédure sera :

```
i=select("tele")
i=shmcmd("@test")
i=select("elodie")
i=shmcmd("@test")
```

Les utilitaires travaillant avec un serveur (voir plus bas : prompter, xdbox, ipcstat, ...) travaillent par défaut avec la clé 1000. Si un autre serveur doit être sélectionné, cela se fait avec l'option `"-k <clé>"`.

1.5 LE CLIENT EST UN SERVEUR

Dans le cas ou le client d'un serveur est lui–même un serveur, il faut prendre garde qu'en tant de serveur, il selectionne toujours son bloc de communication pour y lire les commandes qui lui sont adressée. S'il recoit une

commande de communication, il va l'envoyer sur son propre bloc (justement a cause de la selection courante !) et devenir son propre client (situation blocante). Dans ce cas il est necessaire de lui envoyer les commandes de communication dans une procédure contenant l'ordre de selection du destinataire. Par exemple :

```
sh select("spectro")
sh shmcmdw("entrysh /open")
```

1.6 UTILISATION SIMULTANÉE DE PLUSIEURS GROUPES CLIENT–SERVEUR

Pour permettre plusieurs groupes client–serveur de cohabiter sur une même machine, il faut que chacun d'eux utilise une zone de mémoire partagée individuelle.
La manière de différencier les zones de mémoire partagée est donnée par l'attribution d'une clé (valeur numérique) différentes à chaque groupe. Attention, la clé est un entier positif. La clé par défaut est 1000.
La syntaxe d'appel à Inter devient :

```
inter -mem <clé> [autres options]
```

1.7 CONTRÔLE DES ERREURS EN MODE CLIENT–SERVEUR

Un client récupère un status après chaque communication avec un serveur. Ce status vaut :

1. si une erreur s'est produite sur le serveur (à la compilation ou à l'exécution).
2. Si le serveur a été détruit durant l'exécution d'une commande
3. Si l'exécution de la commande a été interrompue par un <CTRL>-C.
4. Si une attente sur le client a été interrompue par un time-out ou un <CTRL>-C.

Dans tout les cas aucune erreur n'est générée au niveau du client. C'est donc la procédure qui doit gérer ces cas.

1.7.1 Interruption d'un serveur

Si un client détecte une erreur dans un de ces serveurs, il peut décider d'interrompre le déroulement des opérations sur une partie (ou la totalités) des serveurs. Dans ce cas, il envoie un signal d'interruption avec la fonction

SIGNAL() (<CTRL>-C) aux serveurs choisis. Ceux qui sont en cours d'exécution s'arrêteront à la fin de la commande en cours, les autres restant dans leur états d'attente.
Par exemple, pour interrompre les clients "tele" et "guid" on tapera :

```
i=select("tele")
i=signal(2)
i=select("guid")
i=signal(2)
```

1.7.2 Interruption d'un client

Pour interrompre un client, un serveur doit simplement terminer son exécution et signaler une erreur. Cette situation est gérée dans la majorité des cas par Inter d'une manière standard. Dans le cas où une situation d'urgence nécessite la fabrication d'une erreur, on utilise la commande "ERREUR /SET <message>".
Par exemple, si on considère un serveur A qui a un problème (ex : faute durant la compilation), avec en attente sur lui un client B et que ce client B soit en même temps le serveur d'un client C. L'erreur qui survient sur le serveur A va monter de manière automatique vers le client B qui doit la détecter et la faire remonter ver le client C. Le code dans la procédure tournant sur B ressemble à :

```
i=shmcmdw("@procedure_pour_serveur_A")
if i.gt.0 erreur /set "Probleme dans le serveur A"
```

Si le client C qui est en attente sur B a des serveurs XX et YY, et qu'il détecte une erreur, il va interrompre ses 2 serveurs. Le code tournant sur C ressemble à :

```
i=shmcmdw("@procedure_pour_serveur_B")
if i.gt.0 then
  i=select("XX")
  i=signal(2)
  i=select("YY")
  i=signal(2)
  erreur /set "Probleme dans le serveur B, arret general"
endif
```

1.7.3 Interruption d'un client lors d'une attente, Time–Out

Lors d'une interruption survenant lors d'une attente sur un serveur la synchronisation peut être perturbée. Par exemple, le client peut être interrompu à cause d'une time–out alors que le serveur est valide, mais ralenti ou stoppé.
Pour effectuer une resynchronisation automatique, la technique est la suivante :

```
i=shmwack(timeout)
if (i.eq.4) then
    if srvwait() status=shmfree()
    erreur /set "TIME-OUT"
endif
```

La fonction srvwait() indique si le serveur est toujours croché. Cela evite de libérer le serveur si ce n'est pas nécessaire.

1.7.4 Fonctions utilitaires

Quelques fonctions permettent de connaître l'état du système ou de le modifier. Ce sont :

SHMNCNT()
: Retourne le nombre de client en attente sur le serveur courant..

CLEARSV()
: Efface les flags d'erreur et de status ainsi que les code et message d'erreur ainsi que le texte de la commande courante dans le bloc de communication.Cette fonction permet de ne pas récupérer d'erreur trainant d'une précédente commande. Si on pose que l'on part d'une situation stable.

SHMZERO()

Met le semaphore zéro à 1 (reset)..

1.8 UTILITAIRES

Ces utilitaires tournent sous Unix, et ont une interaction avec Inter.

1.8.1 PROMPTER

On peut a tout moment utiliser le programme **prompter** qui se connecte à un Inter travaillant en mode serveur et permet de lui envoyer des commandes. Il permet le rappel des commandes comme dans Inter lorsqu'il travaille en mode interactif.

On sort du prompter avec la commande "bye". Les commandes "quit" ou "exit" terminent Inter.

La syntaxe d'appel est :

```
prompter [-k <clé>]
```

La clé permet de choisir un serveur pour les cas traités dans les sections "Utilisation de Plusieurs Serveur" et "Utilisation Simultanée de Plusieurs Groupes Client-Serveur".

Les commandes suivantes simulent un Inter travaillant en interactif, alors que le passage des commandes se fait par la mémoire partagée.

```
sun> inter -server &
sun> prompter
```

1.8.2 XDSEND

Cet utilitaire permet de lancer une commande à inter depuis un shell ou un script.
La syntaxe est la suivante :

```
xdsend options ...
```

Avec les options suivantes :

-k <**clé**>	La clé permet de choisir un serveur pour les cas traités dans les sections "Utilisation de Plusieurs Serveur" et "Utilisation Simultanée de Plusieurs Groupes Client-Serveur".
-f <**command**>	donne la commande que l'on passe à Inter
-s <**script**>	donne une commande système exécutée apres la commande passée à Inter s'il n'y a pas eu de problème.
-E <**erreur**>	donne une commande système ou le nom d'un script que l'on exécute si une erreur s'est produite dans la commande Inter. Dans ce cas, xdsend ne rend pas la main, ce rôle est laissé à cette commande.
-c	indique que xdsend est un client à part entière et qu'il attend d'avoir la main avant de se créer
-h	indique que xdsend rend la main. Ne rend pas la main si une erreur s'est produite dans la commande Inter.
-v	mode verbose : indique ce que "xdsend" comprend de la ligne de commande.
-e	mode echo : affiche toutes les opérations liées à la communication.
-b	émet un beep s'il y a eu une erreur dans la procédure Inter

Exemple :

```
xdsend -v -e -f "@inigen" -c -s "fix" -h -k 1400
```

1.8.3 PANEL

Cet utilitaire permet de placer Inter en attente de la réponse d'un utilisateur. C'est l'équivalent du *INQUIRE* d'Inter.
La syntaxe est la suivante :

```
panel options ...
```

Avec les options suivantes :

-t <**texte**>	texte de la question
-h <**fichier**> **:b**	nom d'un fichier qui est lu afin de créer le header du panel. L'option "b" indique que le texte est à afficher en bold.
-f <**fichier**> **:b**	nom d'un fichier qui est lu afin de créer le footer du panel. L'option "b" indique que le texte est à afficher en bold.
-label <**label**>	label de la fenêtre
-a	ajuste la taille de la fenêtre à son contenu
-geometry <**width**>**x**<**height**>**+**<**x**>**+**<**y**>	largeur de la fenêtre et position
-x <**x offset**>	position de la question selon X
-y <**y offset**>	position de la question selon Y
-s <**y ncar**>	nombre de caractères pour la réponse
-font <**font**>	nom de la font OpenWindows
...	Options standards OpenWindows (voir man xview)

L'utilisation standard est la suivante :

```
local ans=""
ans=system("panel -t""Voulez vous garder le résultat [o/n]"" -s 80 -a")
```

1.8.4 IPCSTAT

Cet utilitaire permet de monitorer l'état des sémaphores de communications

La syntaxe est la suivante :

```
ipcstat [options ...]
```

avec l'option :

-k <**nom_symbolique**> <**clé**> La clé permet de choisir un serveur pour les cas traités dans les sections "Utilisation de Plusieurs Serveur" et "Utilisation Simultanée de Plusieurs Groupes Client-Serveur". On met autant de fois l'option -k qu'il y a de clés.