Rapport de tp7 pour ssecpd

Master2 SDRP Professeur: Alexandre DENIS

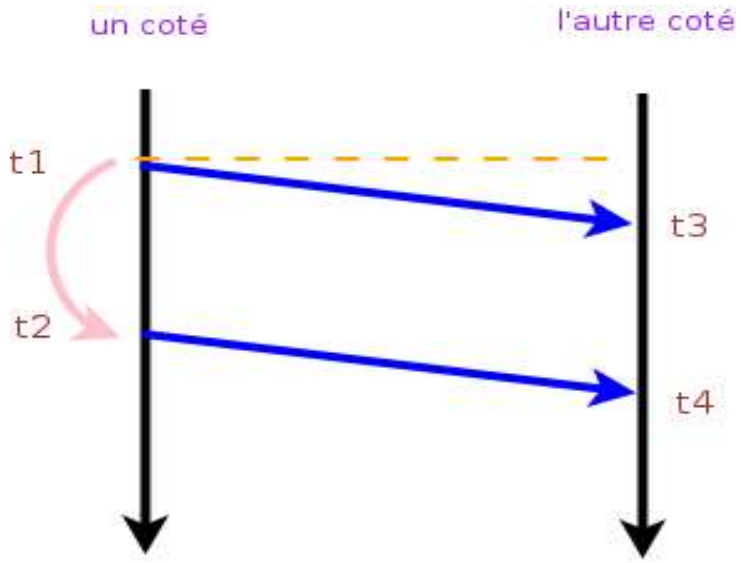
étudiant: LouRuding

****

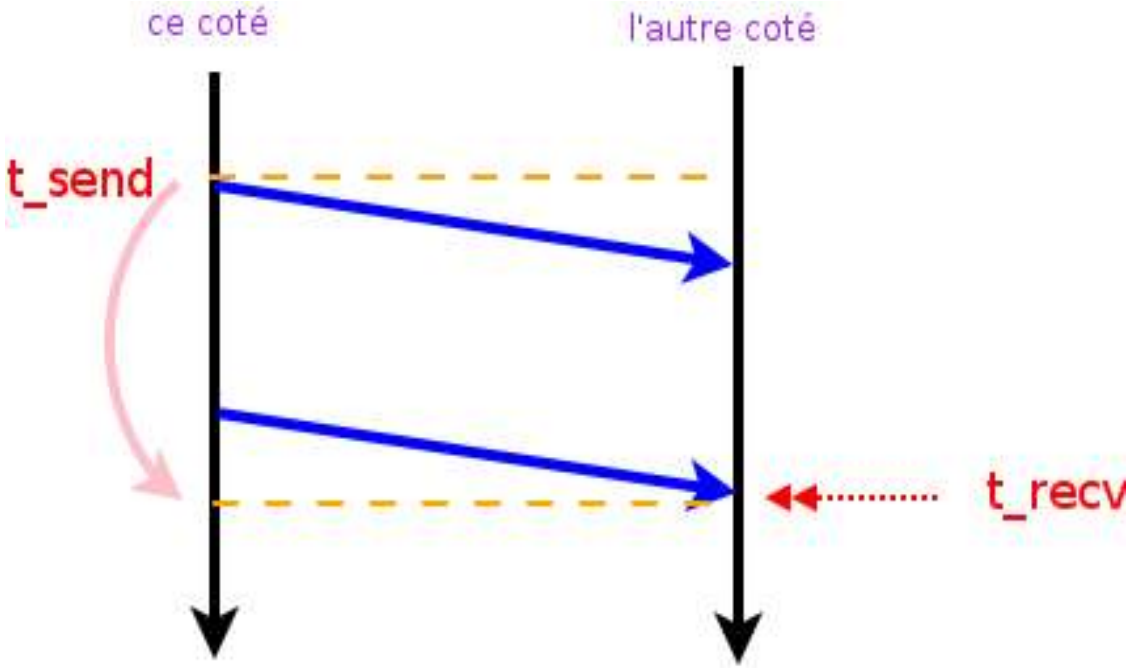
Index

1. Benchmark de référence

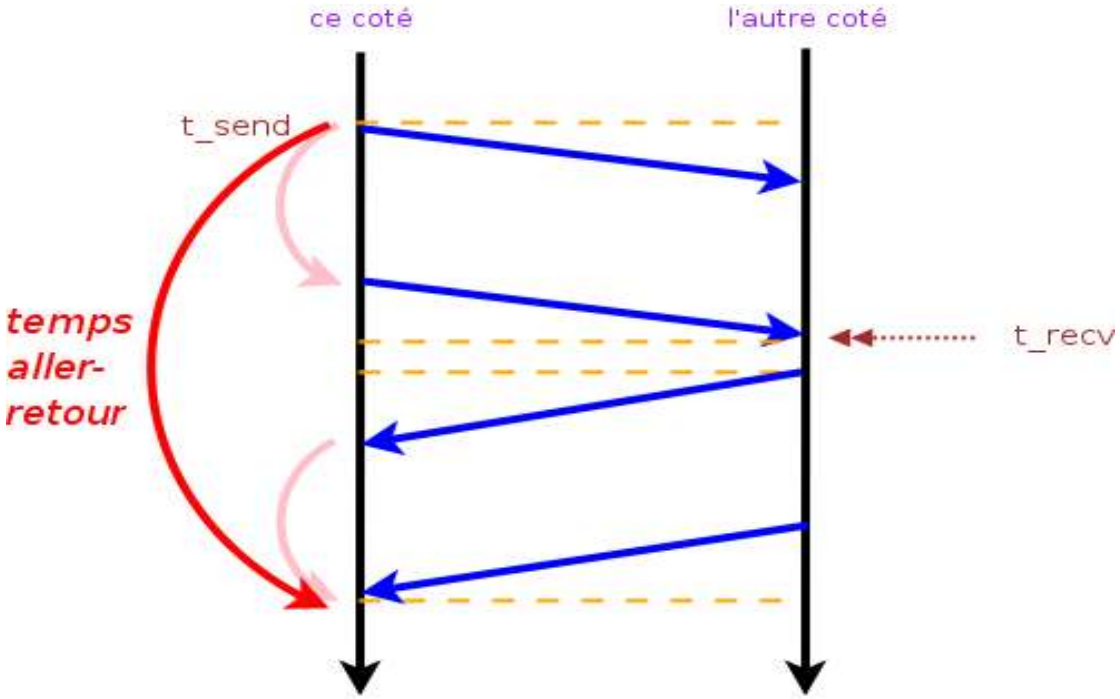
Si on veut mesurer le temps d'émission, on peut directement mettre les deux indice t1 et t2 au tour de mx_isend() + mx_wait(); Donc c'est la mesure de la émission dans le sens unique. En faite le temps (t2 - t1) c'est pas exactement le temps de l'émission complète, c'est-à-dire il y comprend le latence. Voyons la figure en bas, c'est claire que c'est pas ça qu'on veut.



Parce que le vrai temps de l'émission c'est pas t2 - t1; Et aussi si on ne mesure que la coté de recevoir, c'est pas bon non plus, par exemple t3 et t4. En fait la mesure doit être commencée avant de émission d'un coté et terminée après la réception de l'autre coté.



Ça y est , on a trouvé le temps exact qu'on veut. Mais le problème est que nous ne pouvons pas mettre deux tickets t_send dans un coté et t_recv dans l'autre coté. Au moins avec le timing.h on ne peut pas le faire. Donc l'idée c'est que on voit bien un coté émet, l'autre coté reçoit, et si ensuite l'autre coté il renvoie ce que il a reçu exactement à ce coté, on peut presque avoir deux fois de temps de l'émission, parce que les deux coté deviennent la même rôle, et font la même chose. Donc il suffit de mesurer le temps où ce coté commence à envoyer et finit de recevoir.



La figure ci-dessus, si on la plie en deux morceaux, ça divise la graphe en une partie en-haut et une partie en-bas. Donc c'est presque exactement ce que nous voulons. Donc il suffit de diviser le temps d'aller-retour en deux, et c'est le temps d'émission qu'on veut. Et ce que nous mesurons avec cette façons, il y a juste un moment où l'autre coté finit sa réception et avant de commencer l'émission, bon ben ce moment(réactivité) à mon avis sera de moins en moins important si la taille de données transfères sera de plus en plus grande.

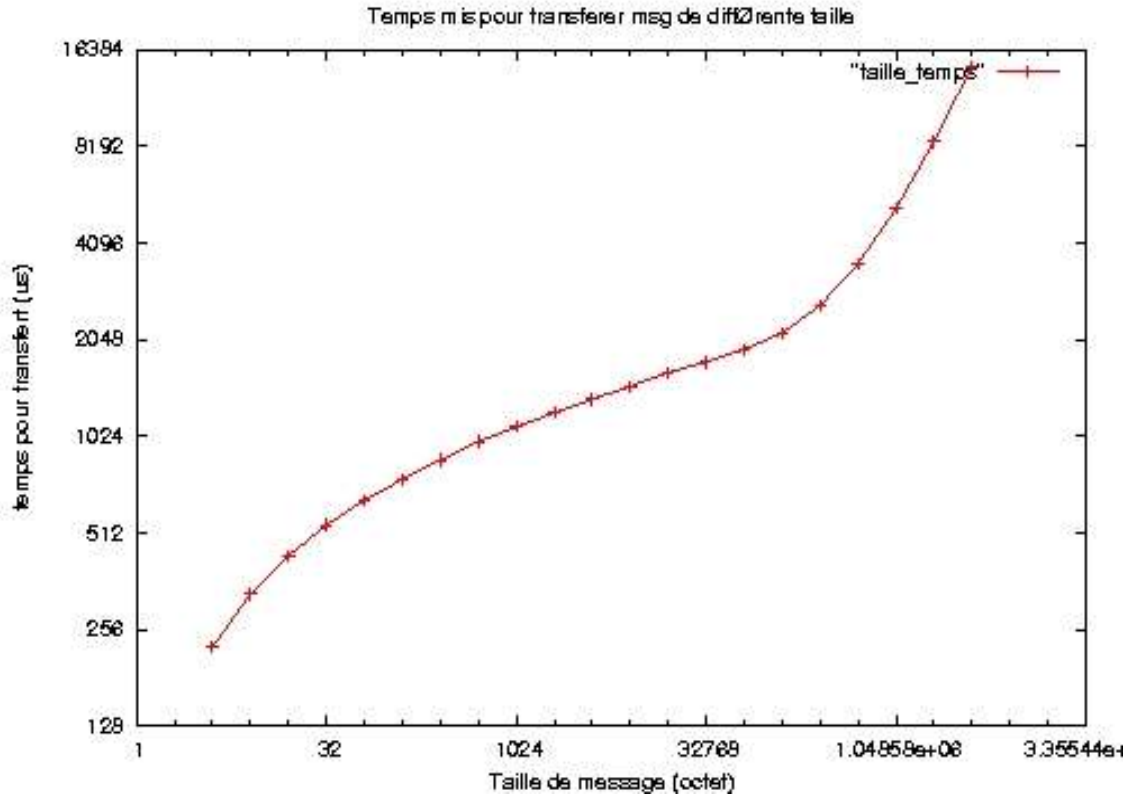
Et donc j'ai fait la benchmark pour la performance en augementant la taille de donnée transférées. Pour tracer le courbe, j'ai fait 10 fois d'allers-retours pour chaque taille.

Voici la courbe taille-temps, et la courbe taille-débit. Et j'ai tester de faire benchmark loacal(sur même machine), et sur deux machine différentes.

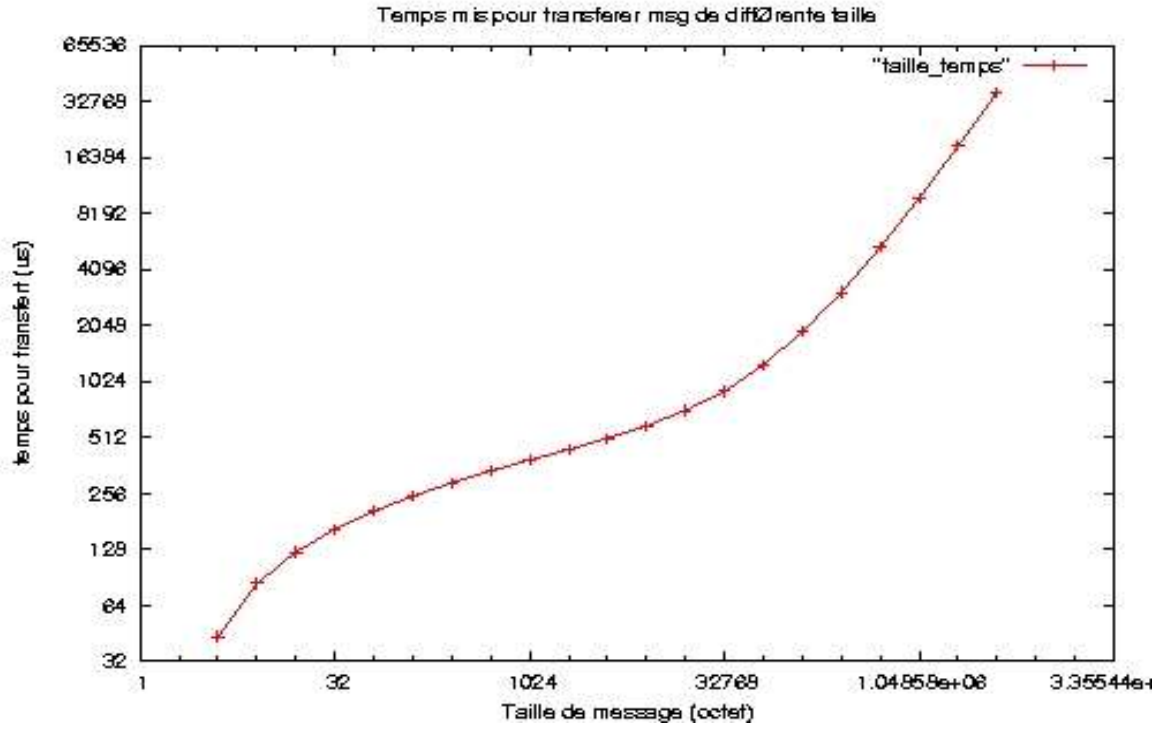
D'abord c'est taille-temps sur une machine, et celle sur deux mahcines .

Veuillez tourner la page suivante.

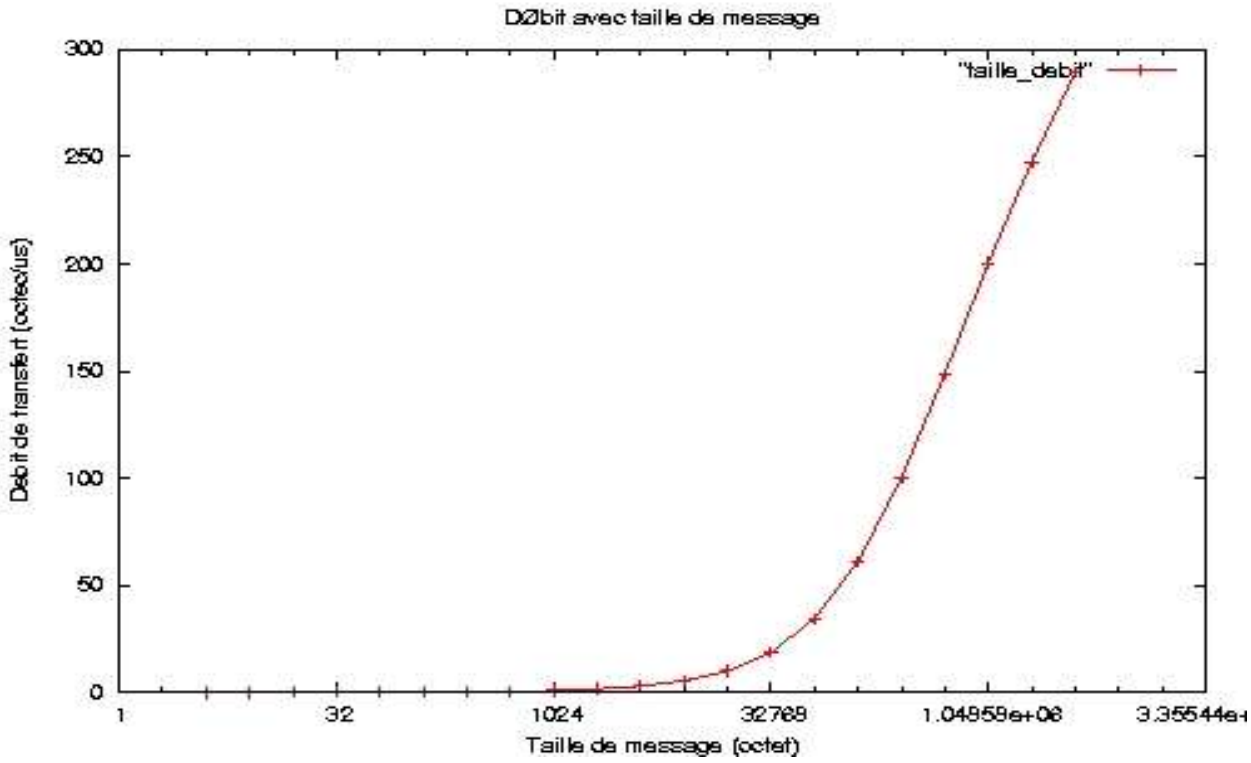
sur une machine



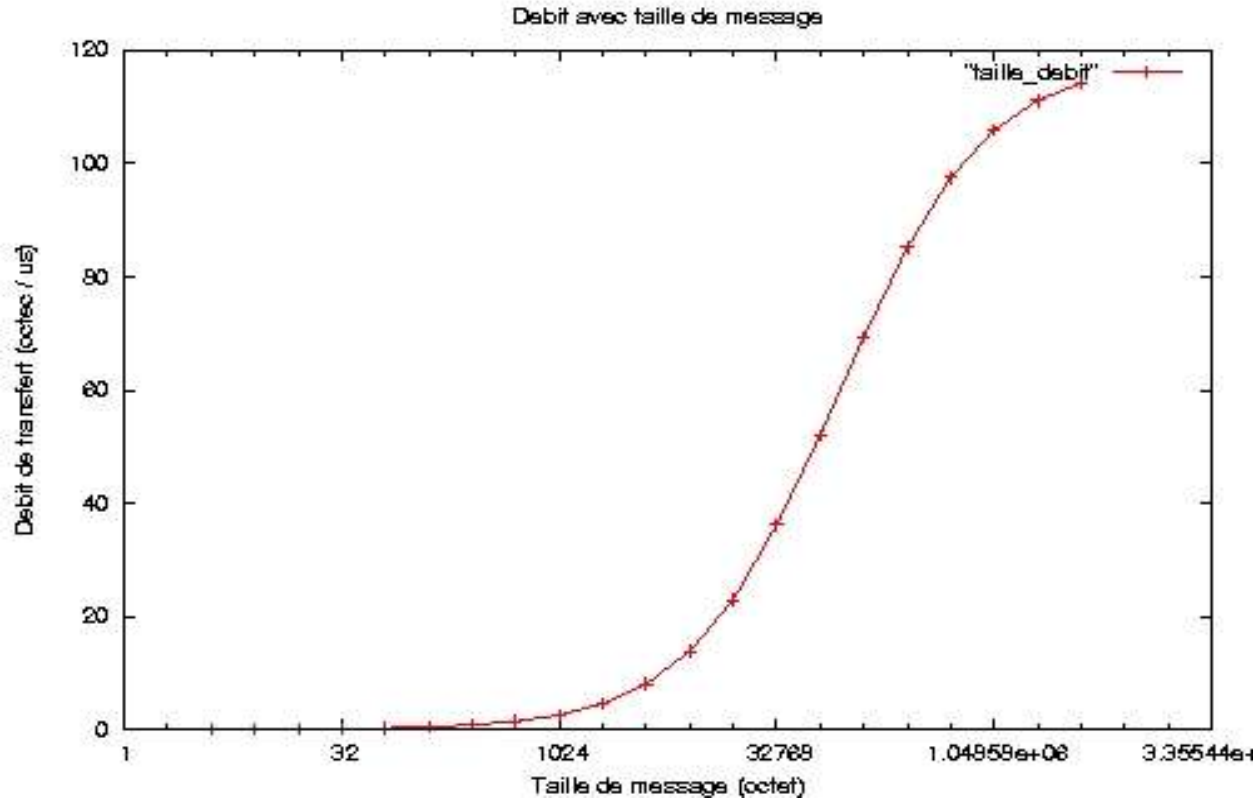
sur deux machines différentes



sur une machine



sur deux machines différentes



D'après ce que j'ai obtenu avec les testes différentes, je trouve que sur machine local, le débit est plus grand, parce qu'il ne compte pas le temps de transfert sur le fibre optique.

J'ai choisi 10 fois d'allers-retours pour chaque taille, parceque je crois que si j'en fait moins de 10 fois, c'est pas assez, il ne peut pas éviter les cas exceptionnel, et si plus de 10, par exemple 50 fois ,100 fois etc ce seront pareil. J'en ai essayé, et les réultats obtenus sont presque la même, donc c'est pas la peine de faire trop de fois .

Sur les figures je vois bien que le débit pour transférer les données de taille inférieur de 8192 et ceux de taille supérieur de 8192 ont grande différence. C'est-à-dire que le débit de transfert de données des taille supérieur de 8192 octets s'augmente beaucoup plus vite que celui pour transférer les données moins de 8192 octets. Et au bout de 1048576 octets l'inclinaison de la courbe est diminuée, c'est-à-dire que entre 8192 octets et 1048576 octets l'inclinaison de la courbe est la plus grande, donc le débit s'augmente le plus vite par rapport que la taille de données que les autres moments. C'est ce que nous avons vu en cours de **SSECPD**, quand les données transférées sont très petites , la latence est très importante et à partir de certaine taille de données la latence devient moins importante, donc le débit s'augmente rapidement. Et aussi jusqu'à certaine taille de données, il peut être plus stable.

2. Multiplexage

2.1 Communications avec plusieurs threads

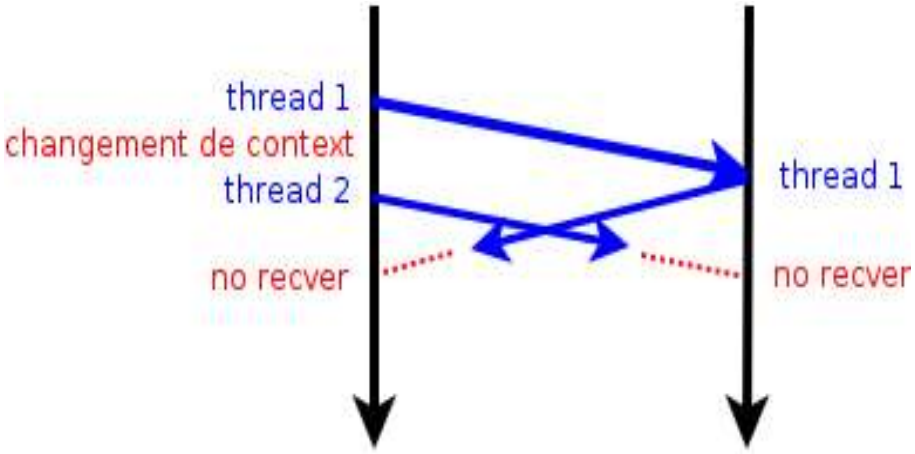
J'ajoute un deuxième thread pour deux cotés, qui veulent se circuler leur propore jeton de différente taille que celui du premier thread entre eux. Donc je peut tout de suite avoir un mauvais résultat qu'il y a une sous-estimation de la taille de buffer. Ce qui est affiché dans le konsole est :

```
knoppix@3[2_Multiplexage]$ ./thread_pong
recepteur demarre
MX:LOU:recv req:req status 5:Data truncated due to undersized buffer
        type: 5 (recv_large)
        state (0x8):
                completed
        mcp_handle : 9
        seg:0xb62049b0,23
        match_mask:(0x00000000_00000001)
        match_val:(0x00000000_00000001)
        matched_val:(0x00000000_00000001)
        slength=29,rlength=23,accum_len=0
        unexpected=0
        local_rdma_id=2
        remote_rdma=-536674302

MX:Aborting
```

La raison c'est que : Quand un thread qui a son propore jeton de la taille plus grande que celui de l'autre, et c'est pas grave qu'il prend le jeton de l'autre. Mais si son jeton est de la taille plus petite que celui de l'autre, il y a une erreur de comme ci-dessus. Parce que quand on fait mx_irecv(... buffer_desc ...), la variable buffer_desc contient un champ segment_length, et il désigne que la taille de données que la fonction mx_irecv() va recevoir, et comme le thread reçoit le jeton plus grand de l'autre, donc le segment_length qu'il désigne ne sera pas assez, et il y peut avoir cette erreur.

Et aussi j'ai essayé de mettre que les threads possèdent leur propres jetons de la même taille pour éviter le problème de différent taille de jeton. J'ai trouvé un autre problème de "deadlock". Je sais pas si le mot va bien ici, le fait est que d'un coté un thread envoie est il est interrompu(pris la main), et l'autre thread envoie aussi. De l'autre coté un thread il reçoit, et il continue d'envoyer, donc deux coté sont en attente d'envoyer. Donc le programme n'avance plus dans ce cas-là. Nous pouvons voir la figure suivante.

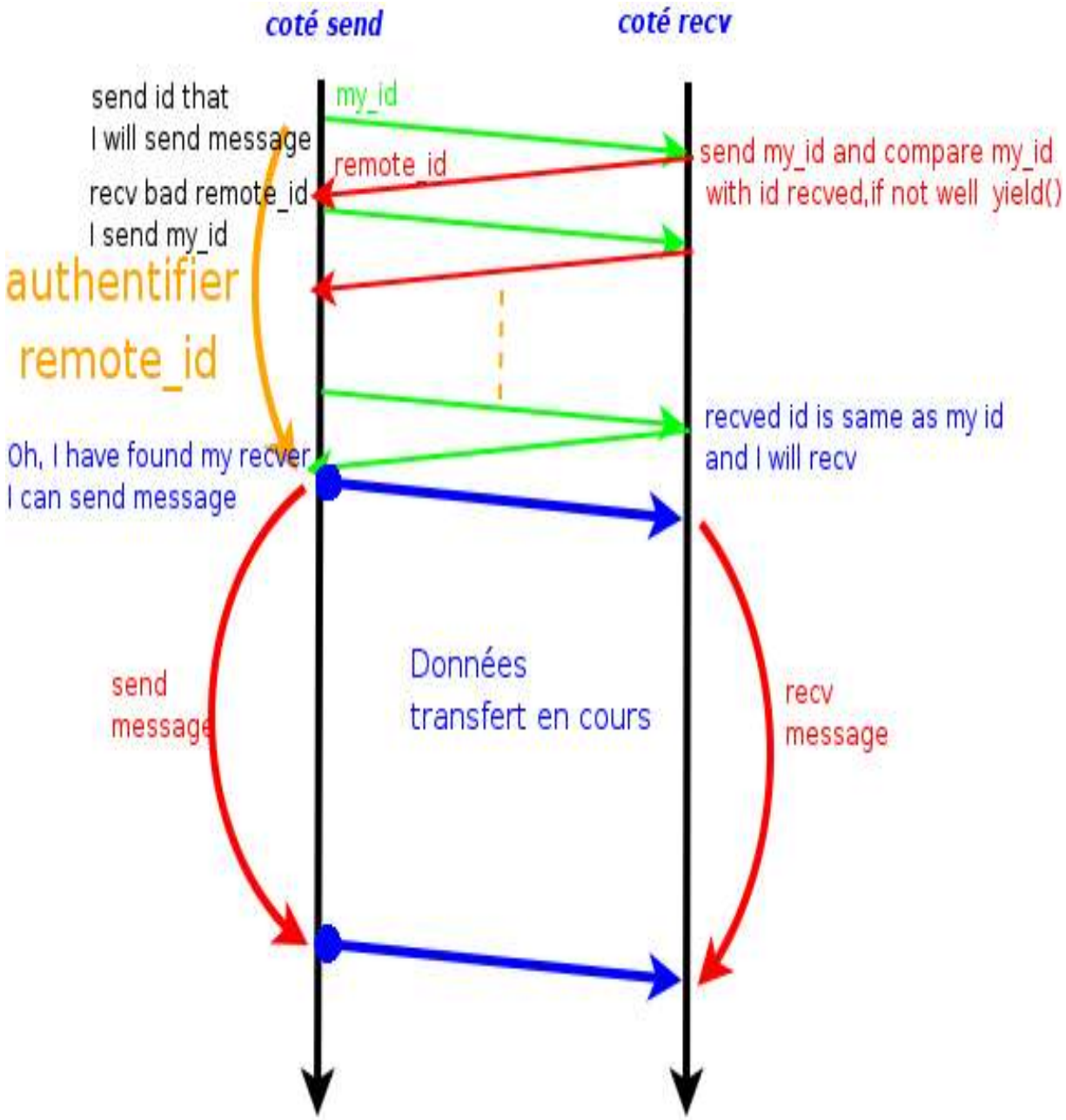


Donc pour résoudre le problème, je crois que on peut utiliser le paramètre match_send et match_recv pour que les différents threads possèdent les différentes clés pour recevoir leurs propres jetons exclusivement. C'est-à-dire que aux deux cotés chaque thread correspondant prend la même match, et différent thread , différent correspondant prennent différents match. Comme ça chaque thread ne peut recevoir que son propre jeton.

2.2 Multiplexage sur MX

Afin de réaliser un mécanique de multiplexage sans utiliser les matchings, j'y ai réfléchit, et je pense que avant de s'échanger les messages, il faut s'authentifier aux deux cotés.

Voyons la figure suivante qui montre mon mécanique pour réaliser multiplexage.

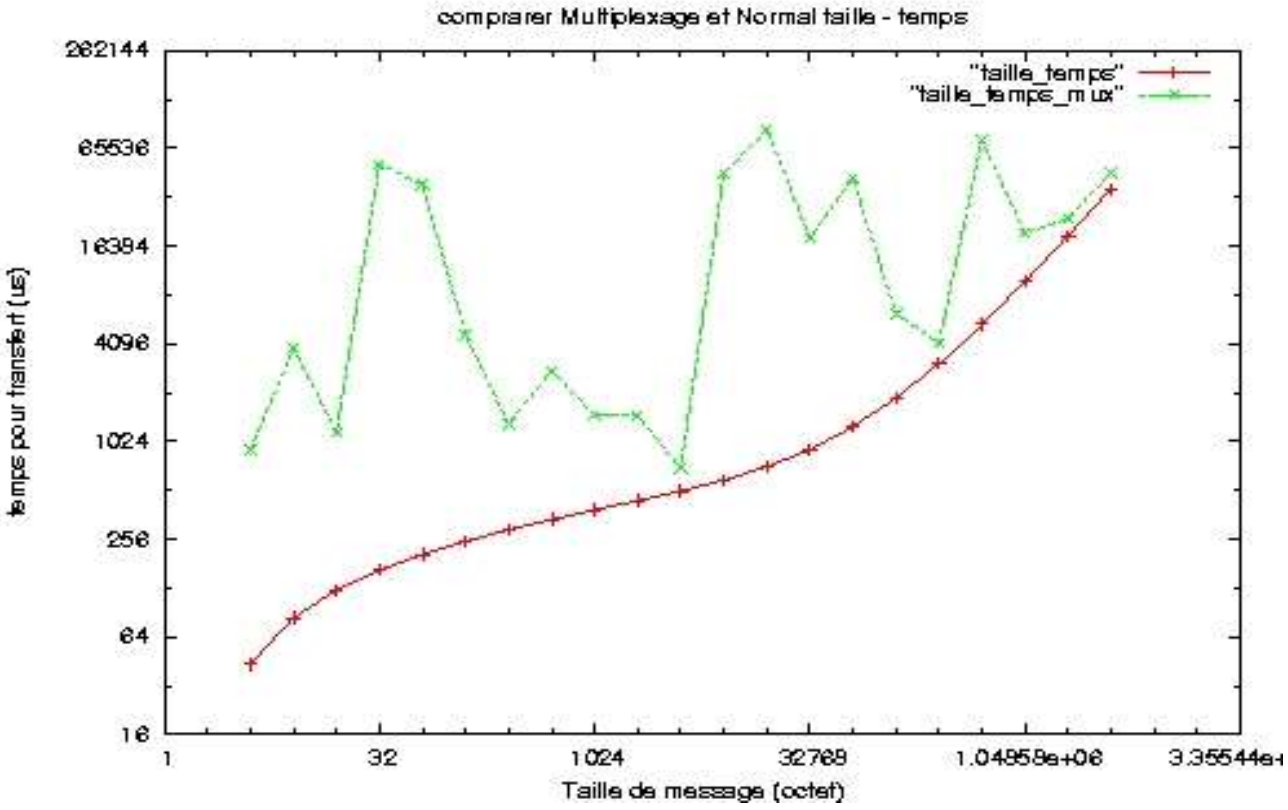


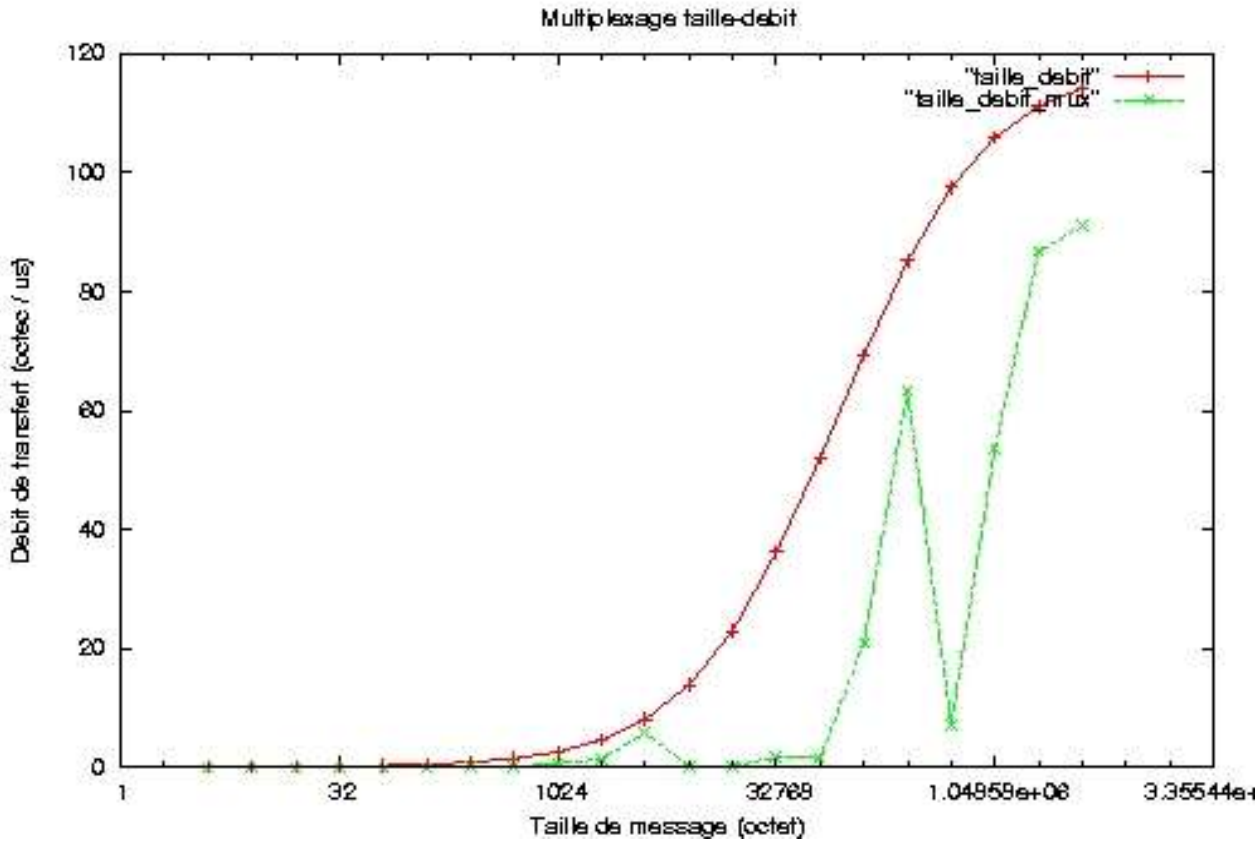
Pour chaque coté, il utilise deux différents endpoint local pour émettre et recevoir. C'est-à-dire, dans un coté, les threads qui envoient les message, occupe un canal logique, et ceux qui reçoit les messages occupent un l'autre canal. Comme il y a quatre endpoint_id pour chaque carte myrinet de notre cas. Donc même sur une machine pour mon implémentation , ça suffit pour deux cotés.

Avec la figure, j'explique mon algorithme pour multiplexage. Donc mon algorithme est que de **coté émetteur**, le thread il envoie numéro de thread à qui il souhaite d'envoyer son message, et il va recevoir un même numéro de l'autre coté. Puis il va vérifier le numéro reçu de l'autre coté, si c'est le bon numéro il peut communiquer toute suite. Sinon il va renvoyer le numéro à qui il veut communiquer jusqu'à il reçoit un bon numéro . Ensuite il peut envoyer le message. Et de **coté récepteur** ,un thread aléatoire vient recevoir un numéro donné par le coté émetteur, et il envoie aussi son numéro au coté émetteur, puis il compare le numéro reçu avec sien, si ce n'est pas la même il abandonne la main. Et d'autre threads prennent la main, ils d'abord vérifier aussi si son numéro est la même que celui reçu. Et si la résultat de vérification est oui, il peut commencer à recevoir le message.

J'ai déjà implémenté ma librairie de multiplexage à 2.1, les threads de chaque coté circulent leurs propres jetons. Et les jetons circulant ne sont plus mélangés. Et après j'ai essayé 10 threads, chaque threads envoient et recoient 10 fois leurs propres jetons avec les correspondants de l'autre cotés. J'ai vérifier que les jetons ne sont pas mélangés non plus.

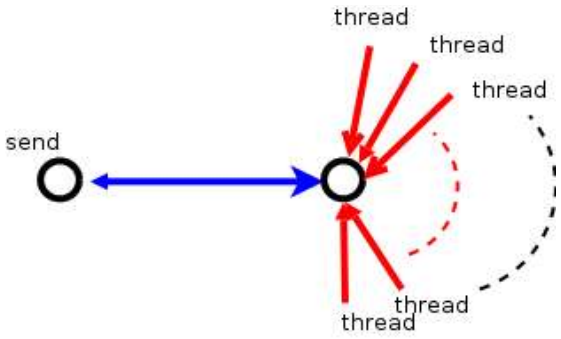
Ensuite j'ai fait un benchmark pour le multiplexage. Donc je crée 10 threads de chaque cotés, et ils font d'un coté émission ensuite réception, et l'autre coté réception puis émission. Donc je calcule le temps moyenne_thread mis par tous les thread. Et pour ce là je vais en faire 10 fois, pour calculer la moyenne. Pour chaque taille. Aussi je divise le temps moyen par 2 pour avoir le temps d'émission de message. Et j'augmente les tailles de 4 octets à 4 M. Donc je trace la courbe avec pour cette benchmark, et aussi j'ai mis la benchmark de Question 1, transfert de données simple.


comprarer Multiplexage et Normal taille - temps
"taille_temps"
"taille_temps_mux"
262144
65536
16384
4096
1024
256
64
16
temps pour transfert (us)
1
32
1024
32768
1.04858e+06
3.35544e+
Taille de message (octet)


Multiplexage taille-debit
"taille_debit"
"taille_debit_mux"
120
100
80
60
40
20
0
Debit de transfert (octec / us)
1
32
1024
32768
1.04858e+06
3.35544e+
Taille de message (octet)

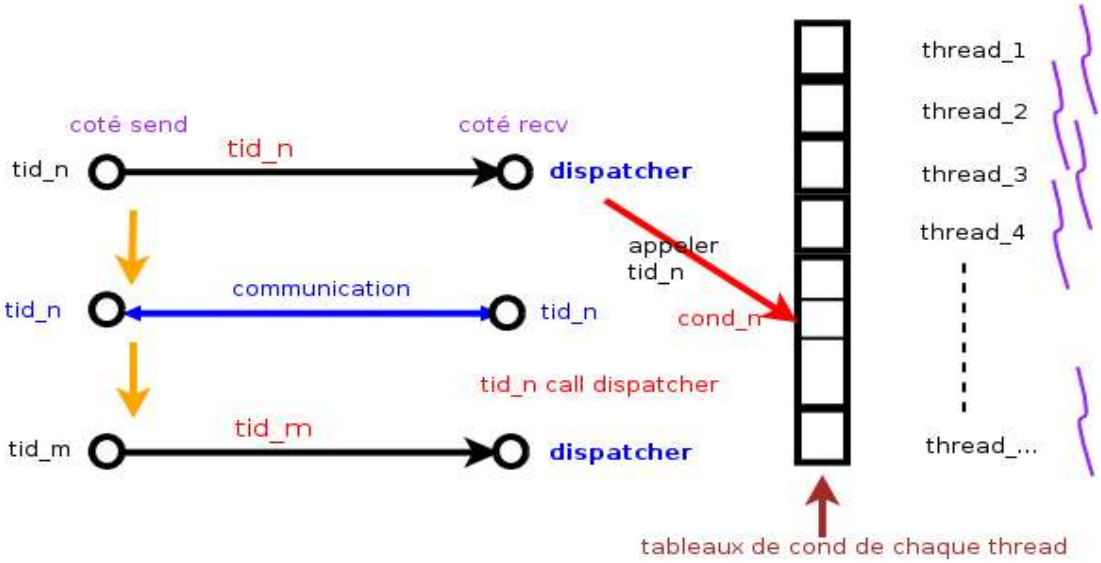
2.3 Performances et optimisations du Multiplexage

Donc en regardant les courbes que j'ai tracées, la performance de multiplexage fait par moi n'est pas du tout bonne. C'est plus réseaux de hautes performances. Le surcoût est de trouver un bonne récepteur avant de chaque émission. Comme il y a 10 threads de chaque coté, donc le cas pire peut être que plus de 10 fois de changement de contextes. Et chaque changement de contexte gaspiller presque 1 us, donc le sur coût est important. Et d'après la courbe on voit bien ,le temps de l'émission n'est pas forcement augmenté graduellement par rapport l'augmentation de taille de donnée. C'est aléatoire, c'est-à-dire la probabilité. En fait c'est la problème de rendez-vous . Pour l'autre coté, un point, et 10 possibilités, donc le probabilité de rendez-vous est 1/10.



Donc la probabilité pour un rendez-vous c'est 1/d(v) (où d(v) est la degré du coté récepteur).

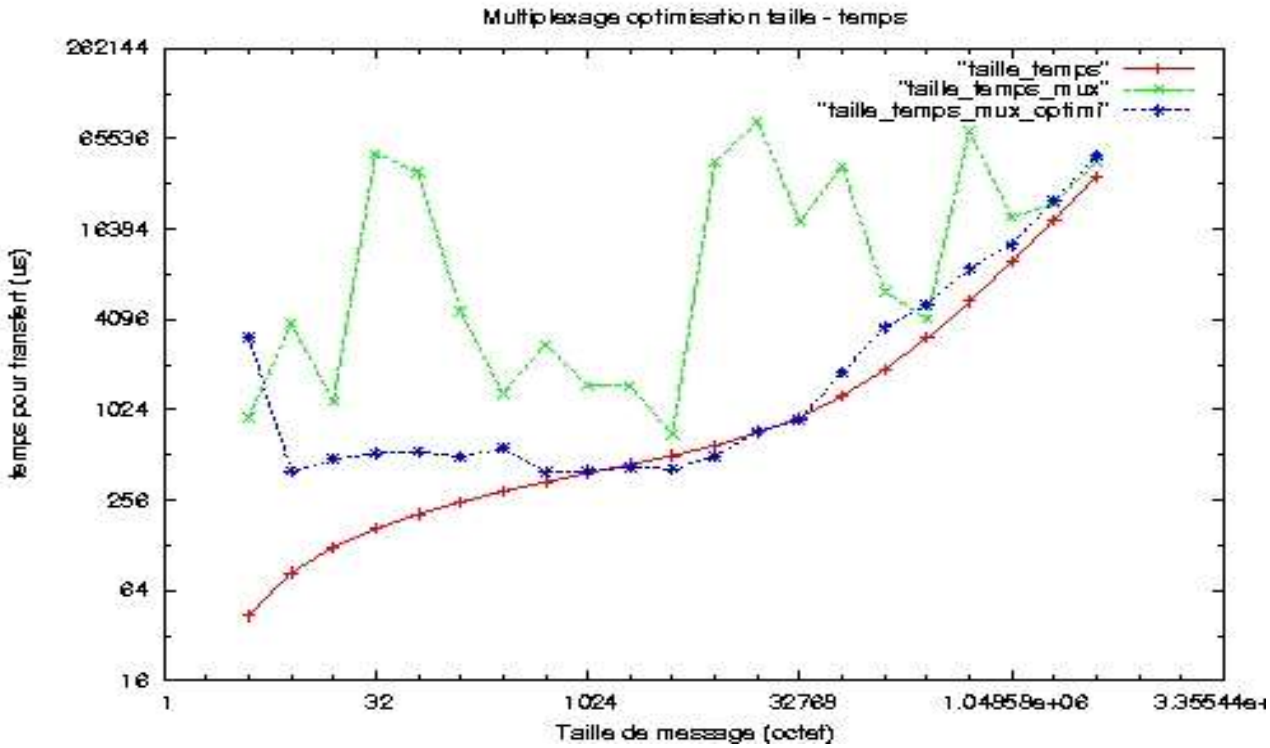
Ce que je pourrai faire pour bien optimiser ma librairie de multiplexage est consisté sur le temps pour le rendez-vous. Ce que j'ai fait c'est chaque thread qui prend la main aléatoirement vient essayer, si ce n'est pas pour lui, il abandonne la main, laisse les autres prennent la main pour essayer. Donc pour le coté émetteur ,il n'y a presque pas grande chose à modifier. Donc pour le coté récepteur ce que je pourrai faire c'est que je crée un thread qui tourne près de coté récepteur pour simplement recevoir le numéro de thread à qui veut parler l'émetteur. Ce thread je peux l'appeler "dispatcher". Donc mon idée c'est d'utiliser le signale cond. c'est-à-dire: chaque thread tiennent une variable de pthread_cond_t propore , une fois un thread appelle mux_recv, il peut se mettre en attente pour son tour avec son propore cond. Donc le thread dispatcher qui s'occupe de la réception de numéro, peut réveiller le thread exactement correspondant grâce à son propre cond. Et le thread peut venir au terminal de récepteur, pour communiquer directement avec émetteur. Et une fois il finit, avant de partir du terminal de récepteur, il peut réveiller le dispatcher pour venir écouter près de terminal. Comme ça pour prendre le rendez-vous, le sur coût est une fois de changement de contexte. Et avec 10 threads ou bien 100 threads ça va énormément diminuer le temps de prendre rendez-vous. Pour expliquer plus précisément je peut faire quelques chemin pour bien expliquer.



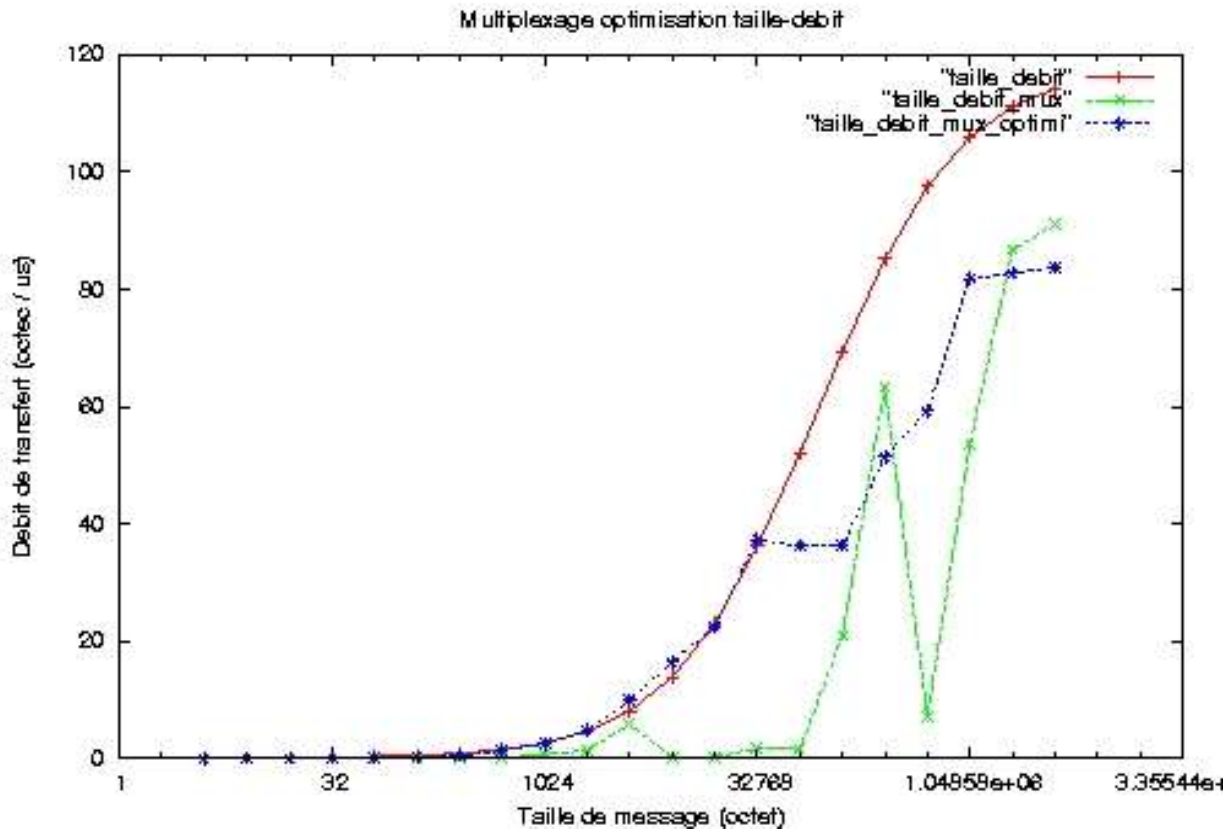
Et voici, les étapes. D'abord, le thread 'dispatcher' écoute le numéro, et il va appeler le thread grâce à son cond avec quoi il s'est mis en attente. Et après il finit communication avec coté émetteur, avant de partir il appeler dispatcher pour venir s'occuper de les appels suivants.

J'ai réaliser ce que j'ai raconté ci-dessus, et je vais aussi refaire le benchmark pour comparer la performance que j'aurai obtenir avec avant.

Voilà la courbe de benchmark pour mesurer la performance de mon implémentation de ma librairie multiplexage.



La figure ci-dessus est celle pour montrer la comparaison entre émission simple(rouge), émission de multiplexage primitive (vert), et émission de multiplexage optimisées(bleu). Elle montre le temps mis pour transférer des données de taille 4 octets jusqu'à 4 Mo. On voit la courbe bleu est presque proche de la courbe rouge. C'est-à-dire, le sur coût introduit par multiplexage est vraiment bien diminué, et la performance n'est plus éloigné que celle de transmission simple. Et en plus il est beaucoup plus stable que le multiplexage primitif, parce que le sur coût pour chaque rendez-vous est uniforme, et sur. C'est-à-dire il ne dépend plus la chance de trouver le récepteur, et pour communiquer chaque récepteur, il ne faut que une fois de changement de contexte. D'après la figure ce que la figure montre, la courbe de multiplexage optimisée ne s'éloigne pas autant que la courbe de transmission mono pour chaque taille. J'y ai réfléchi, et aussi quand je programmais pour optimiser la librairie j'ai rencontré beaucoup de problèmes de mouvements de threads irréguliers. Donc je pense que le surcoût n'est pas simplement la seul fois de changement de contexte, parce que ,des fois quand le thread "dispatcher" reçoit le numéro à qui veut parler l'émetteur, et le thread de ce numéro ne s'est pas mettre en attente, donc le "dispatcher" va attente le thread passe dans le mutex et se met en attente , surtout quand il y a beaucoup de threads, ce cas se passe plus souvent. Nous allons regardons encore la figure pour comparer les trois implémentaion en taille-debit.



Nous voyons bien que la courbe bleu (celle qui représente multiplexage optimisée), sa tendance de direction de monter est presque la même. C'est-à-dire ,quand la courbe rouge commence à monter avec certaine angle, celle bleu monte aussi. Ce n'est plus comme la courbe vert ne pas avancer stablement .

En fait le multiplexage primitif va mieux pour servir à peu de threads, par exemple si on ne crée que deux ou trois threads, avac multiplexage ça va plus vite, parce que entre deux ou trois threads, pthread_yield(), on peut presque tomber sur le thread qu'on veut .

Le multiplexage introduit un sur coût, et c'est pas possible de quasiment éviter. Parce que ça concerne la réactivité de deux cotés quand il y a identification, de distinguer ou traitement de message etc. En fait c'est ce qu'on dit le latence, le latence peut être toujours diminué mais jamais évité. Donc on ne peut jamais transferer avec multiplexage comme tranfert simple directe.
Donc le sur coût n'est pas évitable.

Merci Monsieur.