Programmation Réseau Système

Implémentation d’un serveur TCP

Pierre Ripoll
Jean-Nicolas Winter

Première implémentation

2 processus :

- un père qui va **gérer les connections** sur le port “public”
- les fils qui **gèrent le transfert de fichiers** pour son client dédié, puis une fois fini arrête son processus.

1 problématique :

gestion lors de la connection car il faut que le fils soit **déjà** sur le nouveau port avant que le client ait envoyé son premier message (nom du fichier)

Résolution :

Coordination entre le père et le fils grâce à **des signaux**, et uniquement lorsque le fils est **prêt à écouter** sur le **numéro de port dédié**, le père envoie ce numéro au client

Première implémentation

Fils connecté :

- il possède un **buffer** contenant tout le fichier
- envoie une partie de ce buffer
- il attends de recevoir un **ACK** avec un timeout = RTT
- si on ne reçoit pas l'ACK du dernier paquet durant le timeout, on **renvoie la fenêtre**

Implémentation d'un mécanisme de Slow-Start pour avoir la Fenêtre :

-n'était pas très efficace car on n'arrivait pas à avoir une fenêtre optimisée

Solutions envisagées :

- **Multi threading**
- **Buffer Circulaire**

Optimisation du “transfert de fichier”

Pour le transfert de fichier, on a décidé d’optimiser le fils pour l’envoie et la réception des paquets :

Utilisation de Threads (pour pouvoir faire des actions en parallèle).
Utilisation d’un buffer circulaire, chaque case contient:
numéro du paquet, paquet à envoyer, **durée de validité du paquet** (timeout).

Pour chaque case du buffer circulaire, on a un **Thread** qui **décrémente le “Time”** jusqu’à zéro pour la gestion de RTT

2 Threads :
-send qui vérifie chaque case du buffer circulaire entre deux pointeurs “**start**” et “**stop**” et envoie ceux dont le “Time” est zero
-receive qui écoute le client et met “ack” selon les **acknowledgement reçus** dans le buffer circulaire pour le paquet correspondant et déplace le **pointeur “start”** (indique au *send* qu’il n’a plus besoin d’envoyer ceux reçu)

Main : gère le buffer circulaire = **enlève les paquets** dont on a reçu les “ACK” et **met les nouveaux paquets** à envoyer dans le buffer, puis déplace le **pointeur “stop”** (indique que le *send* peut envoyer d’autres paquets)

Threads et mutex

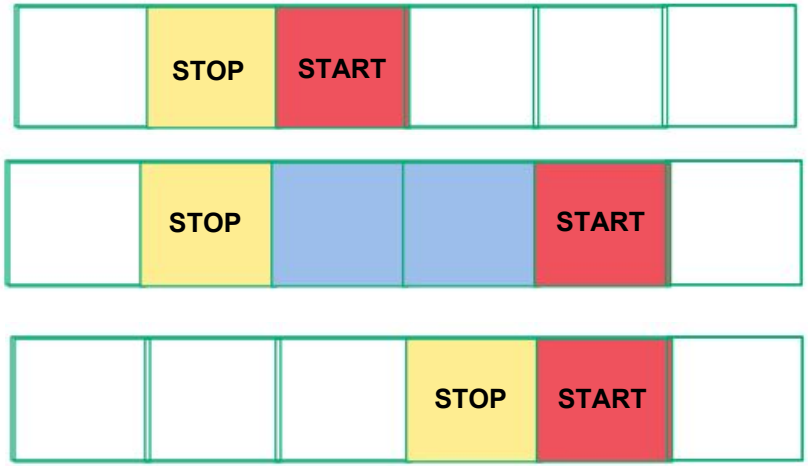
L'implémentation d'un **thread time** pour toutes les cases du buffer est assez **lourde** : les ordinateurs ne sont pas forcément très puissant dans la **gestion des threads**.

Ainsi, lorsque l'on enlève la gestion du **timeout** (envoie en continue de la fenêtre), on économise beaucoup de **ressources**. Cependant, on **inonde** le réseau ce qui peut provoquer **congestion** et **perte de paque**t.

Pour les **mutex**, les enlever permet d'économiser du temps car les threads n'attendent plus la libération de la ressource, mais on se retrouve alors avec des **zones critiques non protégées** menant à des bugs. C'est pourquoi on a privilégié la **fiabilité** de notre serveur en conservant les mutex.

Chargement du buffer

Notre buffer circulaire



Enfin, avec la fonction “fread” on chargeait le **fichier en entier** dans un buffer : problématique si le fichier était trop grand ➜ maintenant on charge petit à petit lorsque l’on prépare un nouveau buffer

Plusieur paramètres qui influence le débit

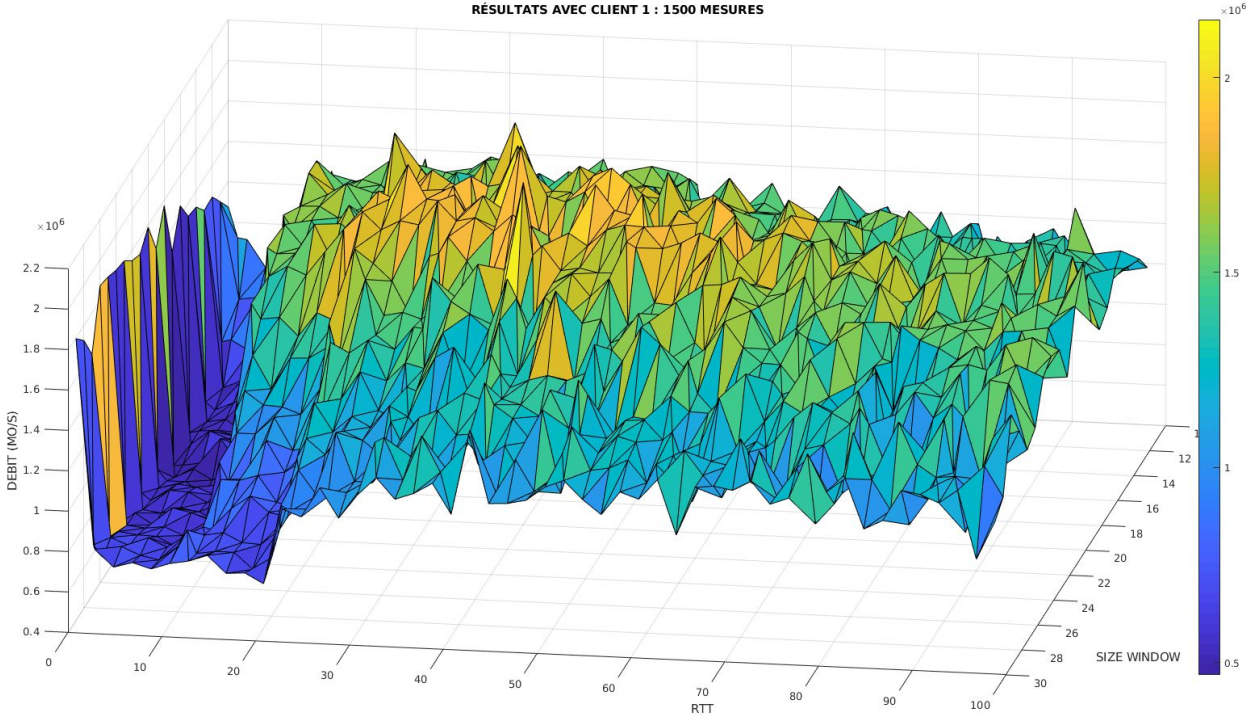
Taille du segment à envoyer : on a une **MTU de 1500,** et le client peut accepter au maximum des segment de 1500, on envoie en tout un **segment de taille maximum 1500.** Ce paramètre ne changera donc pas.

Taille Buffer Circulaire : qui va être la même que la **taille de la fenêtre d'envoi**. (Puisque SNWD fixe)

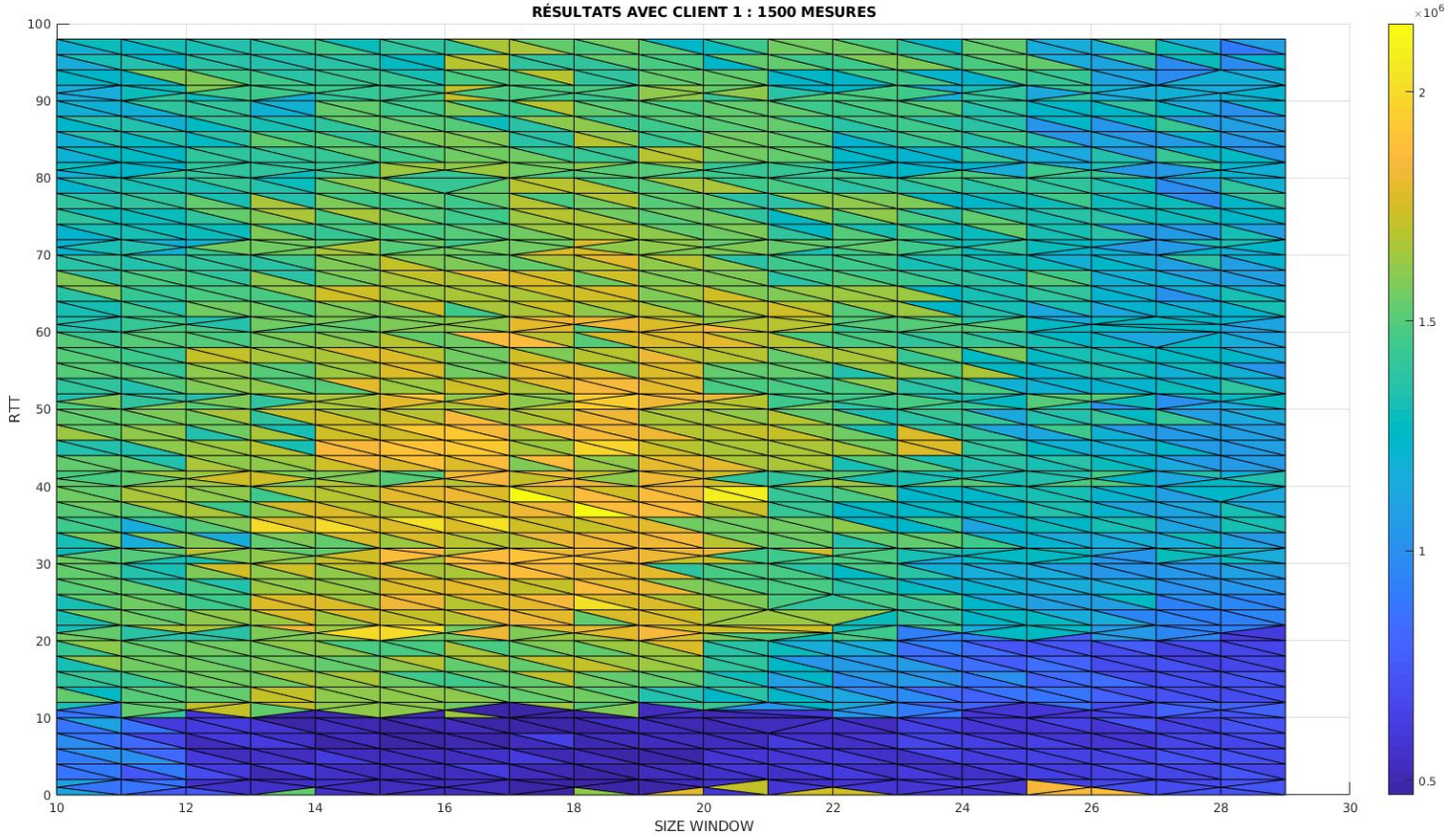
Temps avant retransmission : On peut mettre le **RTT** (temps aller-retour). On pourra l'**évaluer** en fonction du premier paquet transmis. Ici il est paramétré au début.

Fast Retransmit : Transmet le paquet suivant directement après "*x*" ACK reçu de suite. D'après les normes RFC, on a choisit "x = 3"

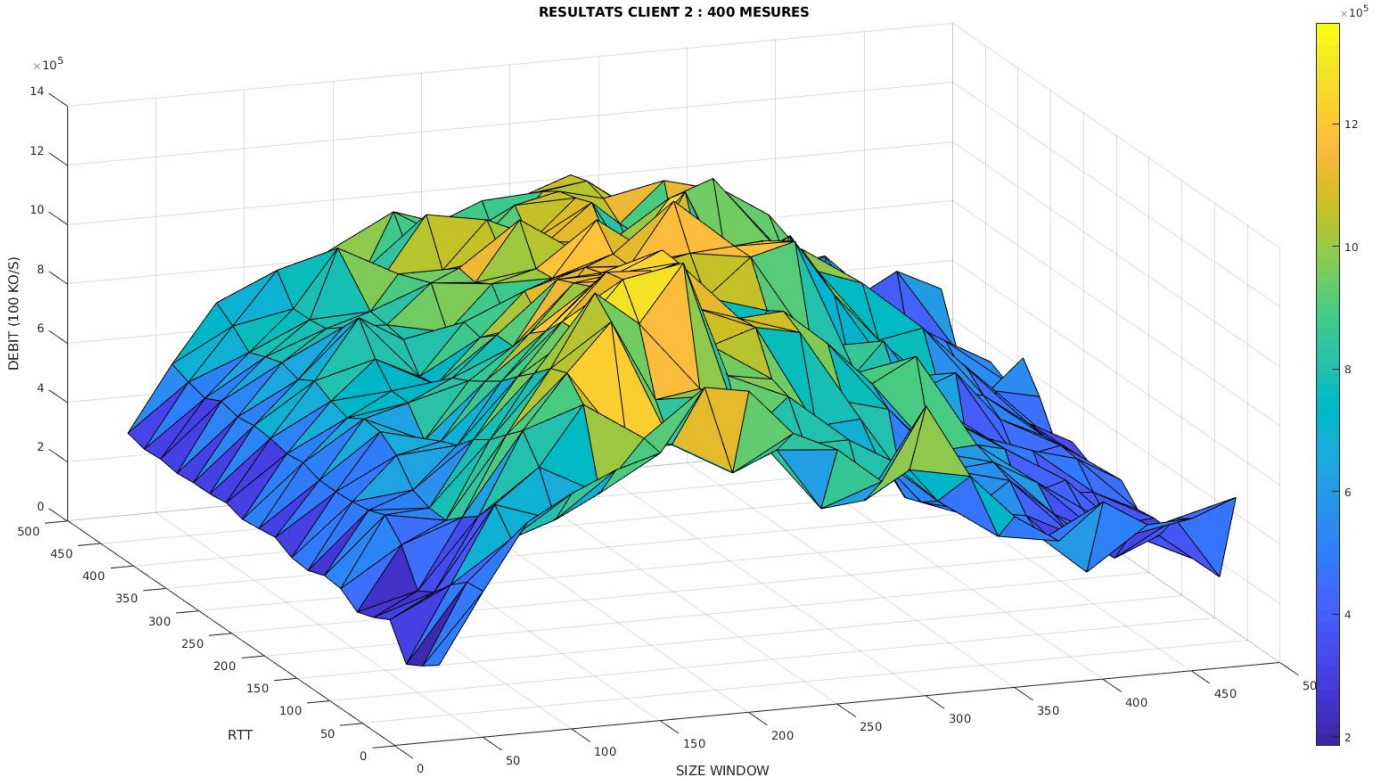
BENCHMARK : Client1



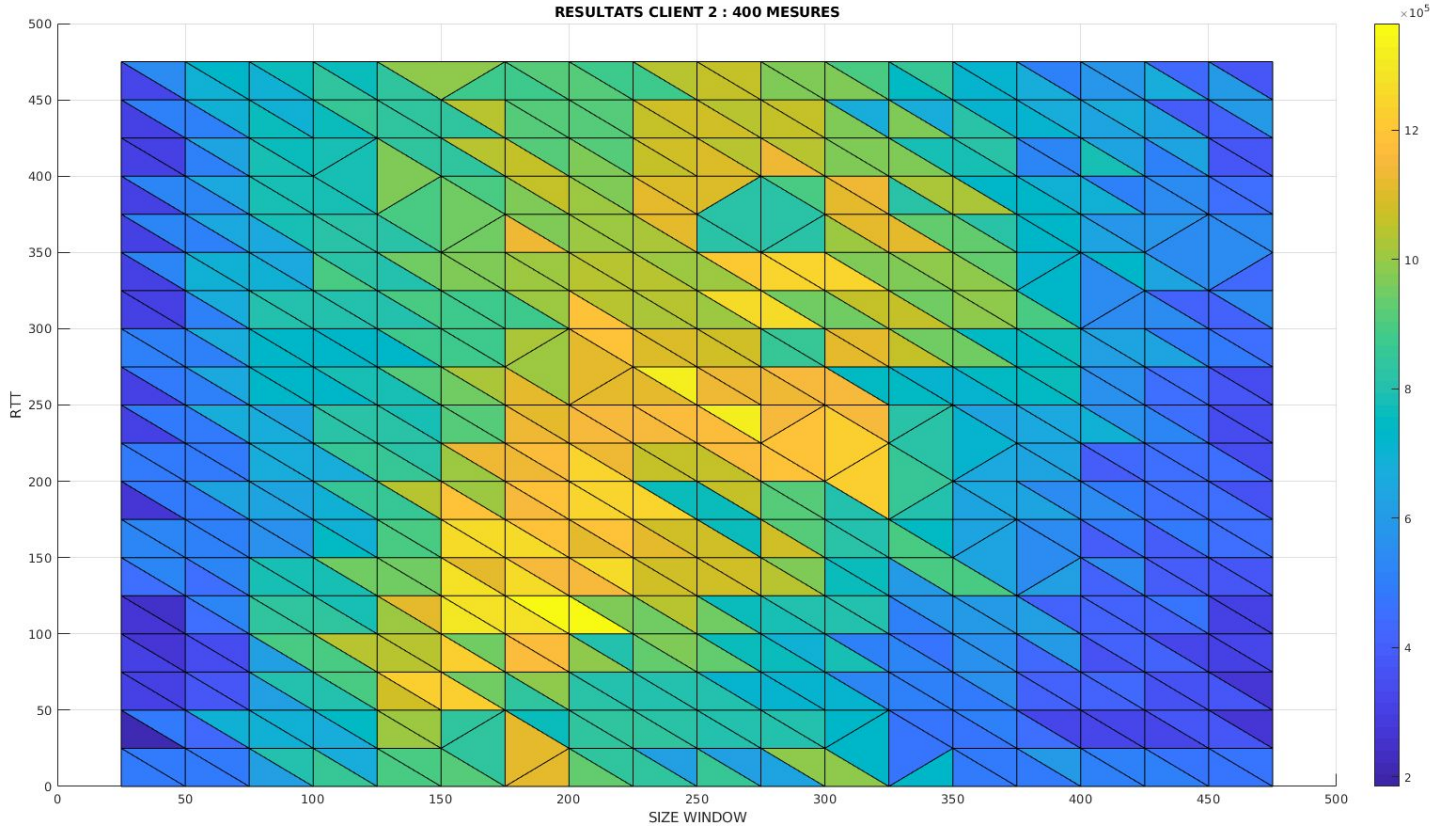
BENCHMARK : Client1



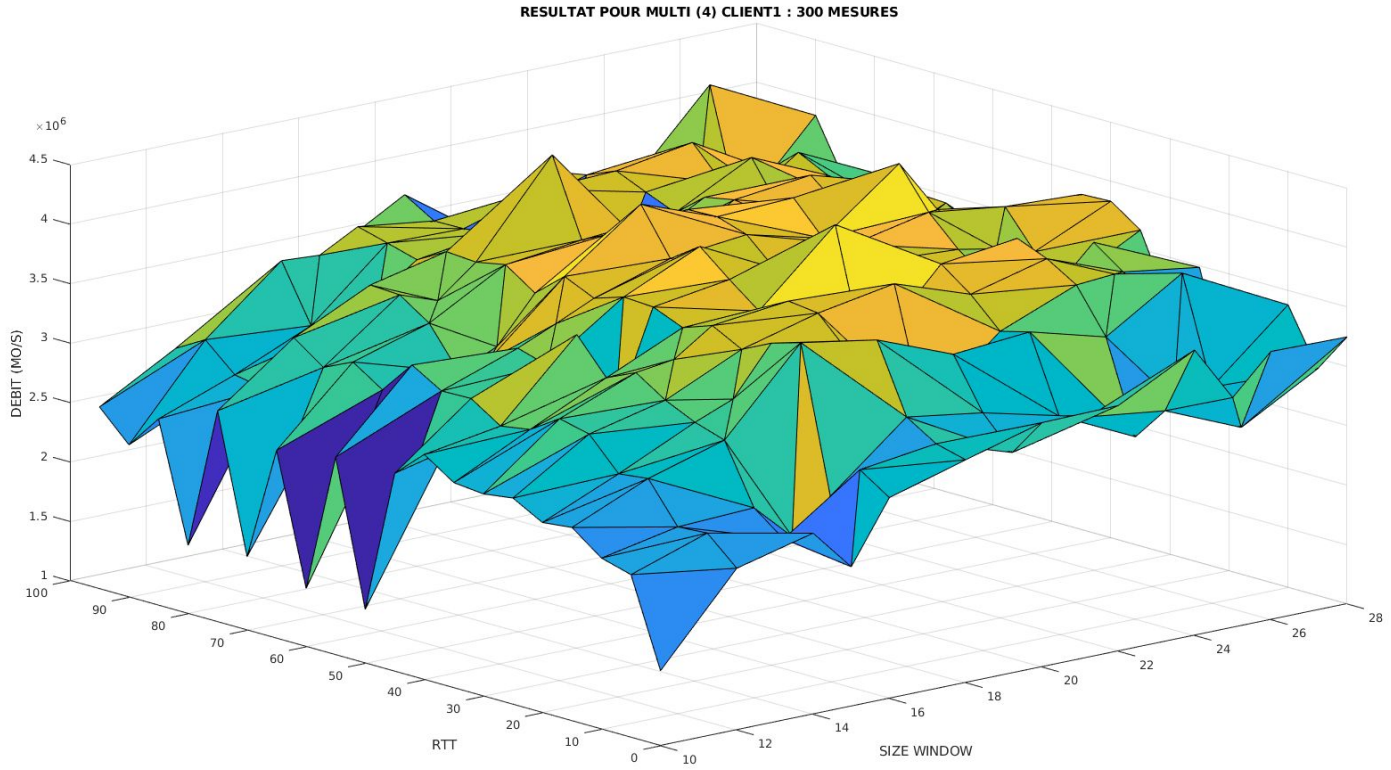
BENCHMARK : Client2



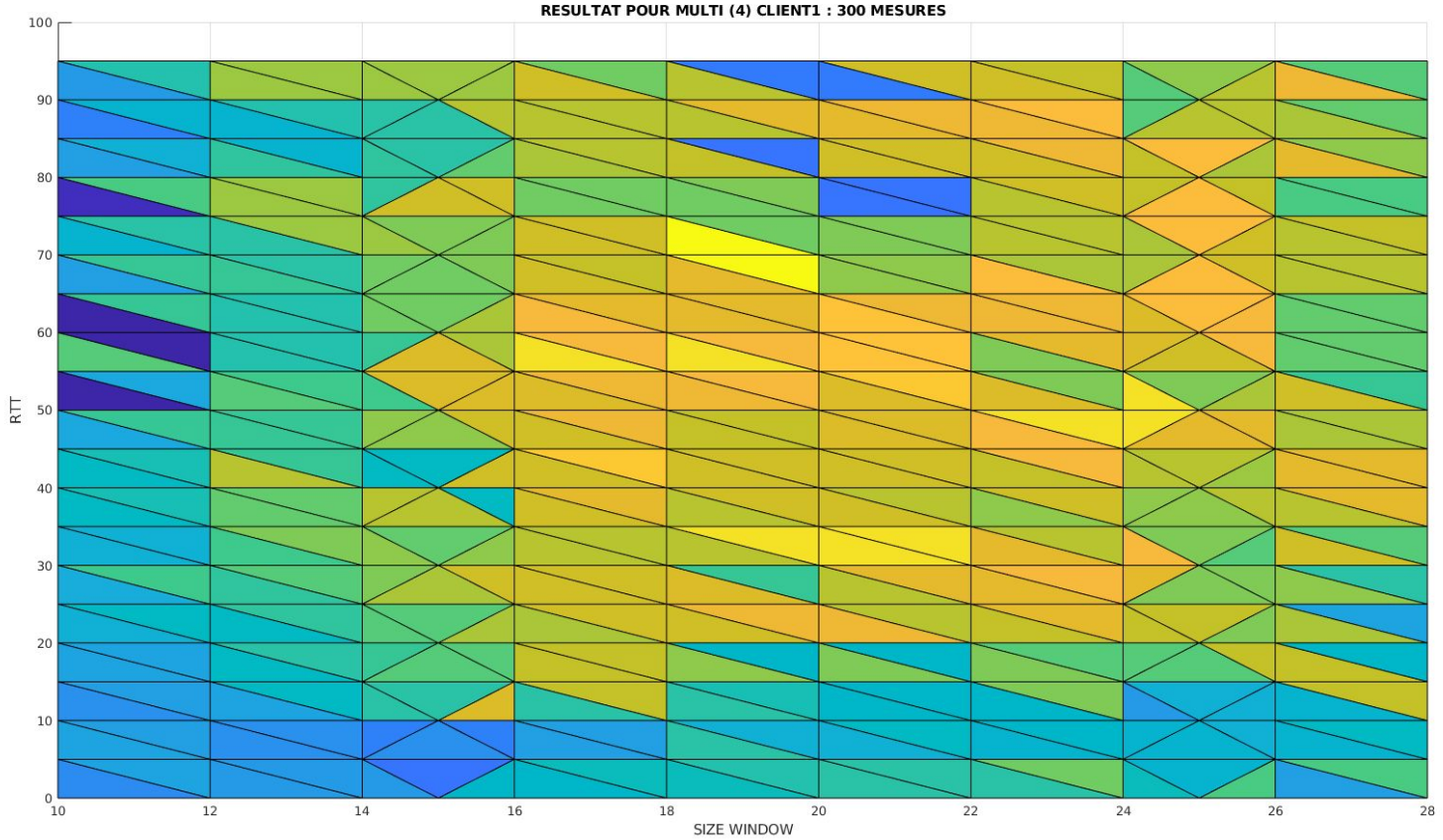
BENCHMARK : Client2



BENCHMARK : Multi-Client (4 client1)



BENCHMARK : Multi-Client (4 client1)



Spécificité :

client1

client2 ➜ énormément d’ACK du même paquet, utilité du Fast-Retransmit, besoin d’une grande fenêtre d’envoi (et donc RTT assez grand)

multiclient

Implémentations à faire

Slow Start :

Objectif : avoir rapidement la taille de la fenêtre d'envoi optimale, on a essayer sur notre première version, cela n'a pas été concluant et on ne l'a même pas essayé sur notre dernière version.

Congestion avoidance :

Permet d'ajuster la fenêtre de façon linéaire, à l'approche d'une valeur élevée de taille de fenêtre

New Fast Retransmit :

Lors d'un Fast Retransmit, on s'attend à recevoir un ACK de toute la fenêtre lorsque le client a reçu le paquet que l'on vient de retransmettre. Dans notre cas, nous pourrions par exemple remettre le "timewait" de toute la fenêtre à RTT car il ne faut pas les renvoyer tout de suite.