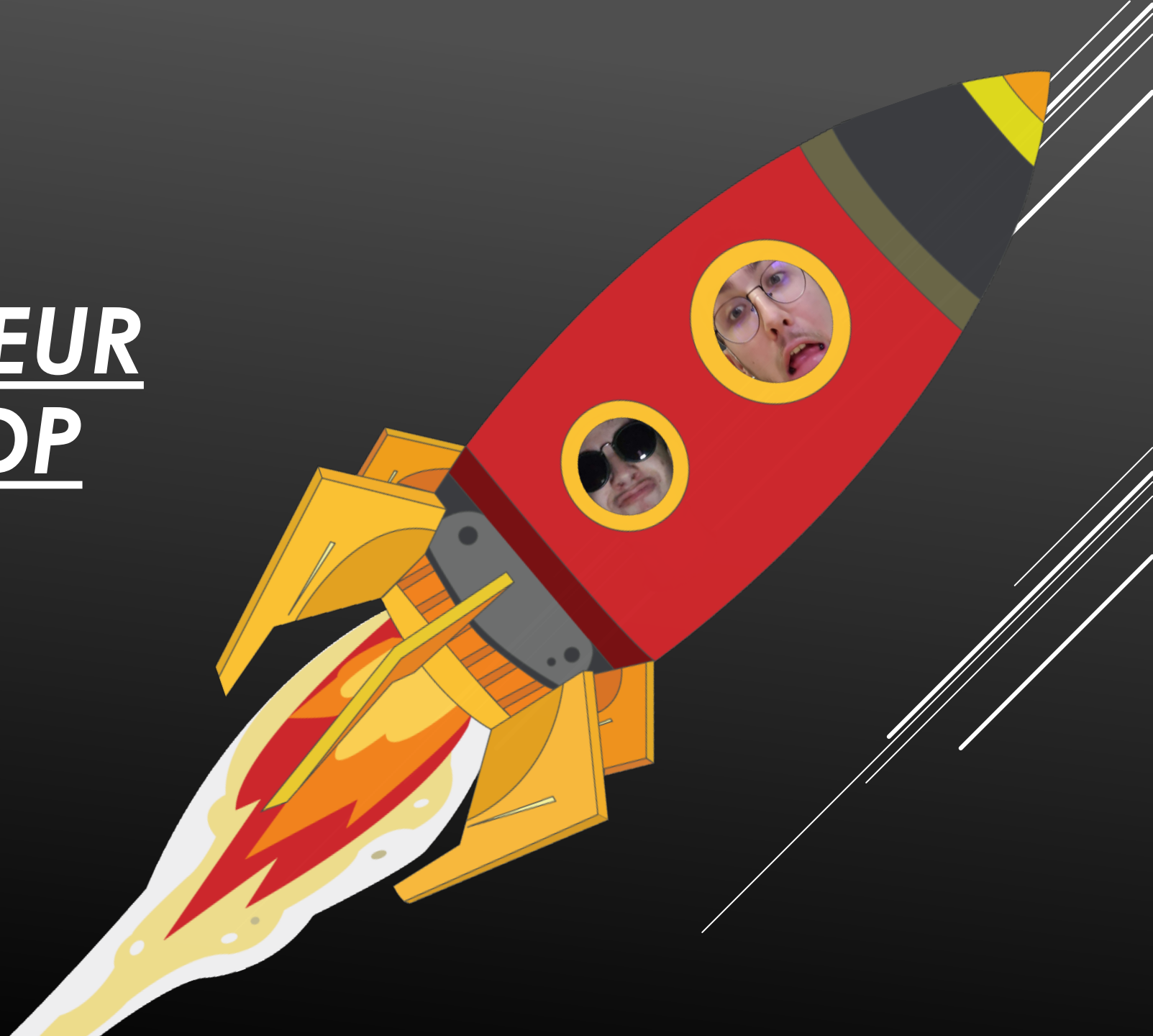
PROJET PRS

CLIENT SERVEUR TCP OVER UDP



DETECTEUR_ML

Mathis Faucheux

Lucas Dufour

PLAN

1. Introduction
2. Choix initiaux
3. Axes de développements
4. Résultats et conclusions
5. Problèmes rencontrés
6. Améliorations futures

1. INTRODUCTION

- Implémenter les mécanismes de TCP sur le protocole UDP :

  Objectifs :

  Protocole
  - Fiable
  - Rapide
  - Flexible

- Deux clients déjà compilés (trois scénarios à étudier)

  => Ecrire les serveurs permettant d'envoyer aux clients un fichier

Le troisième scénario utilise plusieurs clients 1

2. CHOIX INITIAUX

- Ecriture en langage C
  - Plus d'expérience (car TP)
  - Plus bas niveau, contrôle direct sur la donnée
  - Rapide

- Utilisation d'un type « fait maison » pour stocker les data (segments et acks)

  = les Stacks

LIFO pour faire un minimum d'appels mémoires quand on cherche à accéder au dernier ack

3. AXES DE DEVELOPPEMENT

A. Réalisation de la structure de serveur :

- Three way handshake (ouverture à terme d'une socket pour la data)

  = connexion client/serveur
- Segmentation et stockage de l'information dans les Stacks
- Envoi du fichier par segments (contrôle des paquets perdus pour fiabilité)
- Mécanismes de contrôle des fenêtres pour l'envoi (rapidité et efficacité)

B. Changements majeurs pour l'amélioration du débit

C. Recherche d'optimisation des facteurs et constantes internes

CONNEXION CLIENT/SERVEUR

Première socket pour « three way handshake »

Deuxième socket ouverte par le serveur pour échanger l'information utile

⇒ Function handle_syn

```c
/**
 * @brief This function is used to handle the SYN process. It's called when the server receives a SYN packet. It will send a SYNACK packet to the client.
 * This SYNACK packet will contain the port number of the server with the given format: "SYN-ACK<port>". The port number must be between 1000 and 9999.
 * The function creates a new UDP listener and bind it to the new port number.
 * When the SYN-ACK packed had been sent, the server will wait for the client to send an ACK packet. It returns the new socket file descriptor.
 *
 * @param sockfd The socket file descriptor.
 * @param client_addr The client's address.
 * @param client_addr_len The length of the client's address.
 *
 * @return The new socket file descriptor.
 */
int handle_syn(int sockfd, struct sockaddr_in *client_addr, socklen_t client_addr_len) {
    int new_sockfd;
```


SEGMENTATION ET STOCKAGE DE L'INFORMATION DANS LES STACKS

Stockage dans un stack :

- Les segments envoyés (N° de segment, Taille du segment)
- Les acks reçus (N° ACK, Nombre de duplication, RTT)

- Len(Stack) < MAX_STACK_LEN (ici 5, donc 4 éléments maximum)

- Exemple de la structure :
  - (A) : 426 (2) 48 - 426 (1) 67 - 426 (0) 51 - 422 (4) 256

ENVOI DU FICHIER PAR SEGMENTS

(CONTRÔLE DES PAQUETS PERDUS POUR FIABILITÉ)

Algorithme simple pour déterminer le segment à envoyer

Comparaison du dernier ack reçu et du dernier segment envoyé.

⇒ Soit on renvoie le segment

⇒ Soit on renvoie le ack

```c
/**
 * @brief This function is used to give the next packet to send. It implement the fast retransmission mechanism.
 *
 * @param last_ack The last ACK received
 * @param last_seq The last sequence number sent
 * @param timeout this is a flag to indicate if the timeout occured
 * @param eof notify if the end of file is reached
 *
 * @return int The next sequence number to send
 */
int next_seq_to_send(STACK acks, STACK segs, int timedout, int eof) {

    int last_ack = acks->element;
    int last_seg = segs->element;

    if(eof == 1) {
        return last_ack+1;
    }
    if(timedout == 1) {
        if(last_ack < last_seg) {
            return last_ack+1;
        }
        else {
            return last_seg+1;
        }
        //return last_seg +1 ; //last_seg +1 ; //what if ACK + 1 ? -> perf --
    }

    if (acks->duplicate > MAX_DUPLICATE_ACK && last_seg != last_ack +1) {
       // printf("Too many duplicate ACKs, sending back the segment n° %d...\n", last_ack+1);
        return last_ack + 1;
    } else {
        if(last_ack > last_seg && last_seg >= 1) {
            //printf("Duplicate ACK resolved, sending the next segment n° %d...\n", last_ack+1);
            return last_ack + 1;
        } else {
            return last_seg + 1;
        }
    }
}
```

MÉCANISMES DE CONTRÔLE DES FENÊTRES POUR L'ENVOI (RAPIDITÉ ET EFFICACITÉ)

Choix d'une **fenêtre statique** : double à chaque envoi réussis (1 seg puis 2 puis 4 etc)

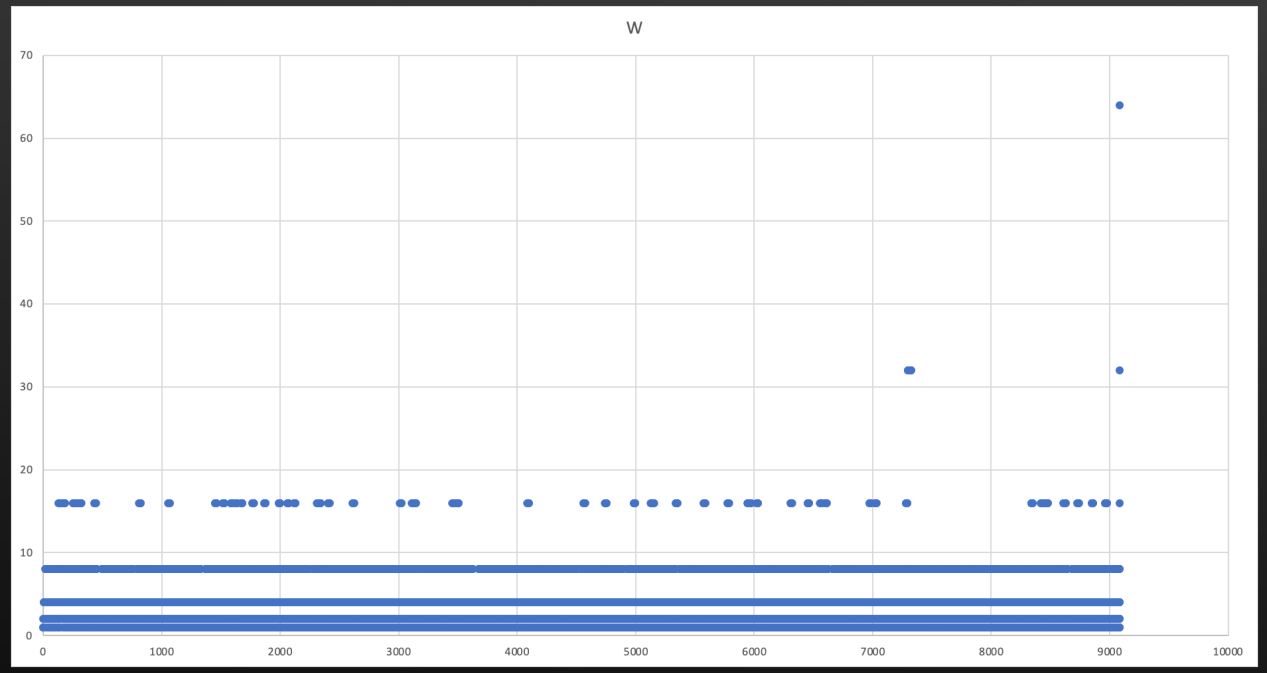
Implémentation :

- Slow start (la fenêtre double)
- Congestion avoidance (pas très efficace)
- Fast retransmit

Implémenté mais pas activé :

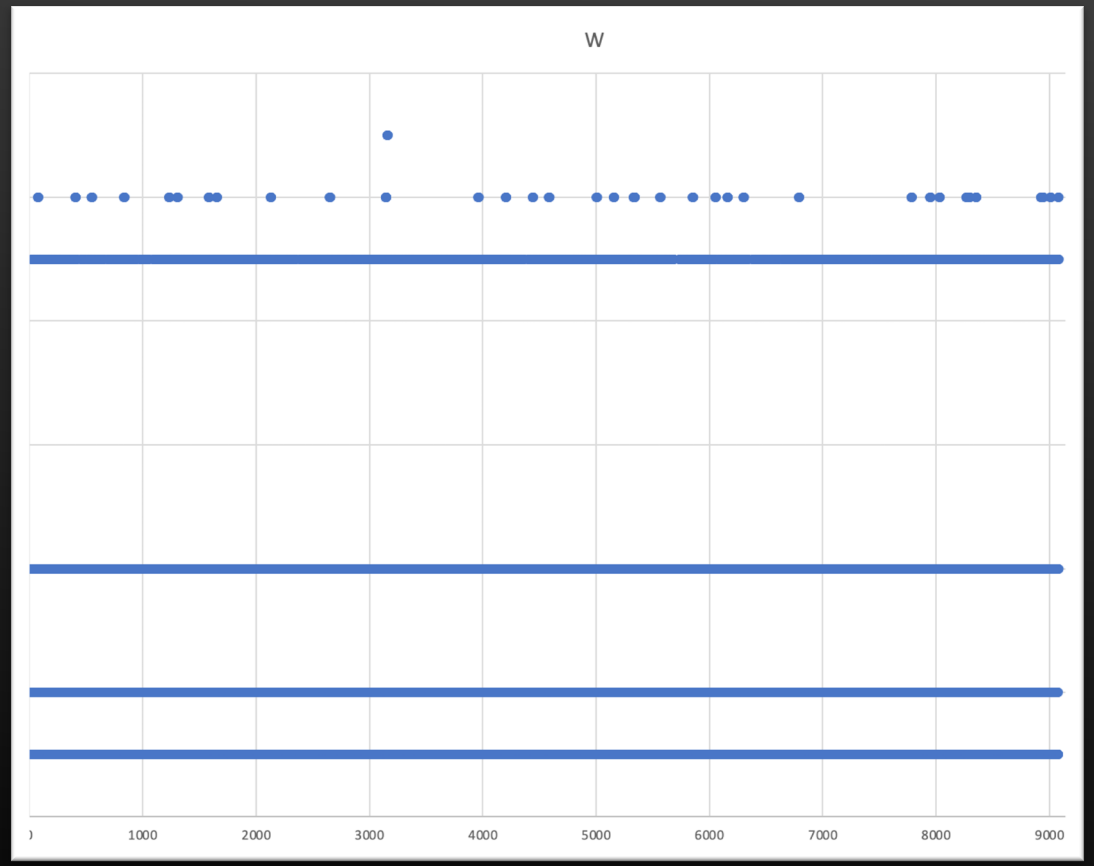
- Fast recovery

OPTIMISATION DE LA TAILLE DE FENÊTRE CLIENT 1

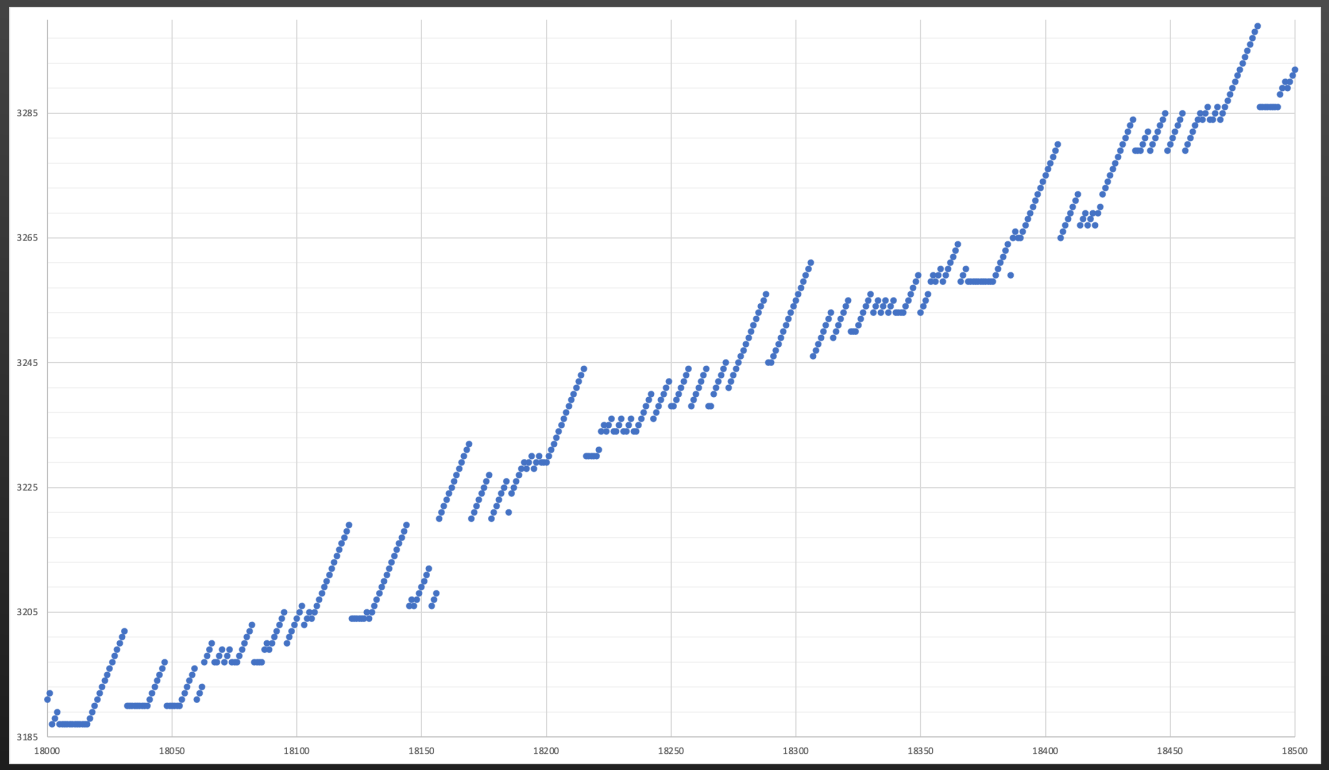


- Slow_start_thresh = 1024
- Debit ~ 1MB/s

OPTIMISATION DE LA TAILLE DE FENÊTRE CLIENT 1



- Slow_start_thresh = 8
- Debit ~ 950kB/s



Envoi des paquets 3185 à 3285.

Envoi par fenêtres (les paquets se suivent).

Abscisse : numéro de l'envoi
ex : le paquet 3205 est le 18000ème envoi

CHANGEMENTS MAJEURS POUR L'AMÉLIORATION DU DÉBIT

Voici une liste des changements qui ont drastiquement amélioré notre débit

- Implémentation de fast retransmit
- Implémentation de l'estimation du timeout : 10kB/s → 100kB/s
- Implémentation des fenêtres : 100kB/s → 400 kB/s
- Changement de la taille du stack (de 10 à 4) : 400kB/s → 550 kB/s
- Stack des Ack en gardant que le dernier ACK (600 kB/s -> 1000kB/s)
  - Car moins de 'bug' sur les fenêtres

RECHERCHE D'OPTIMISATION DES FACTEURS ET CONSTANTES INTERNES

On cherche à optimiser nos variables internes pour les adapter à chaque clients

CLIENT 1

DEFAULT_RTT 2000 us
ALPHA 0.01
BETA 0.05

CLIENT 2

DEFAULT_RTT 3000 us
ALPHA 0.01
BETA 0.05

```c
/**
 * @brief This function is used to estimate the timeout based on the RTT.
 *
 * @param rtt The last mesured rtt
 *
 * @return int The new timeout
 */
int estimate_timeout(double rtt) {
    sRtt = (1 - ALPHA) * sRtt + ALPHA*rtt;
    devRtt = (1 - BETA) * devRtt + BETA * fabs(rtt - sRtt);
    //printf("Estimated RTT: %f us\n", sRtt);
    return (int) (sRtt + 4 * devRtt);
}
```


RECHERCHE D'OPTIMISATION DES FACTEURS ET CONSTANTES INTERNES

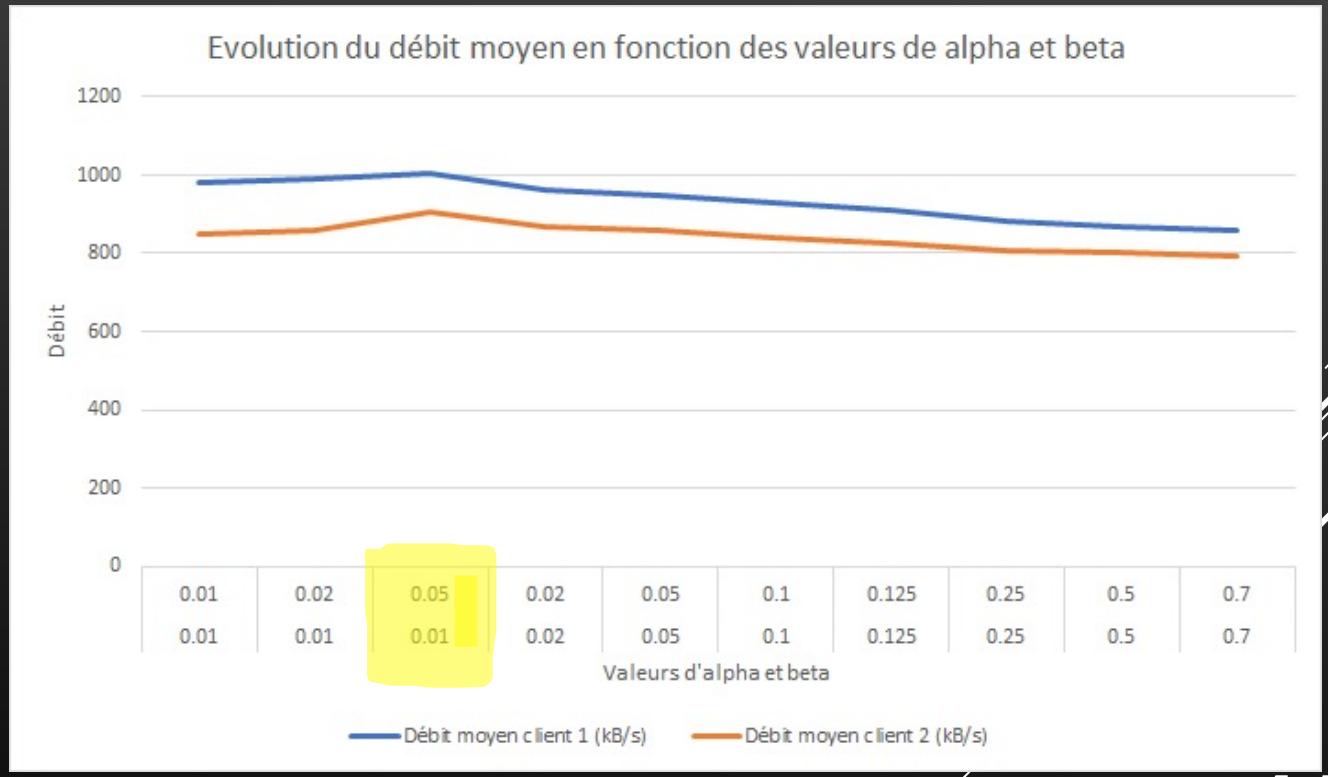
On cherche à optimiser nos variables internes pour les adapter à chaque clients

CLIENT 1

SRTT 500µs
DEFAULT_RTT 2000 µs
ALPHA 0.01
BETA 0.05

CLIENT 2

SRTT 500µs
DEFAULT_RTT 3000 µs
ALPHA 0.01
BETA 0.05



RECHERCHE D'OPTIMISATION DES FACTEURS ET CONSTANTES INTERNES

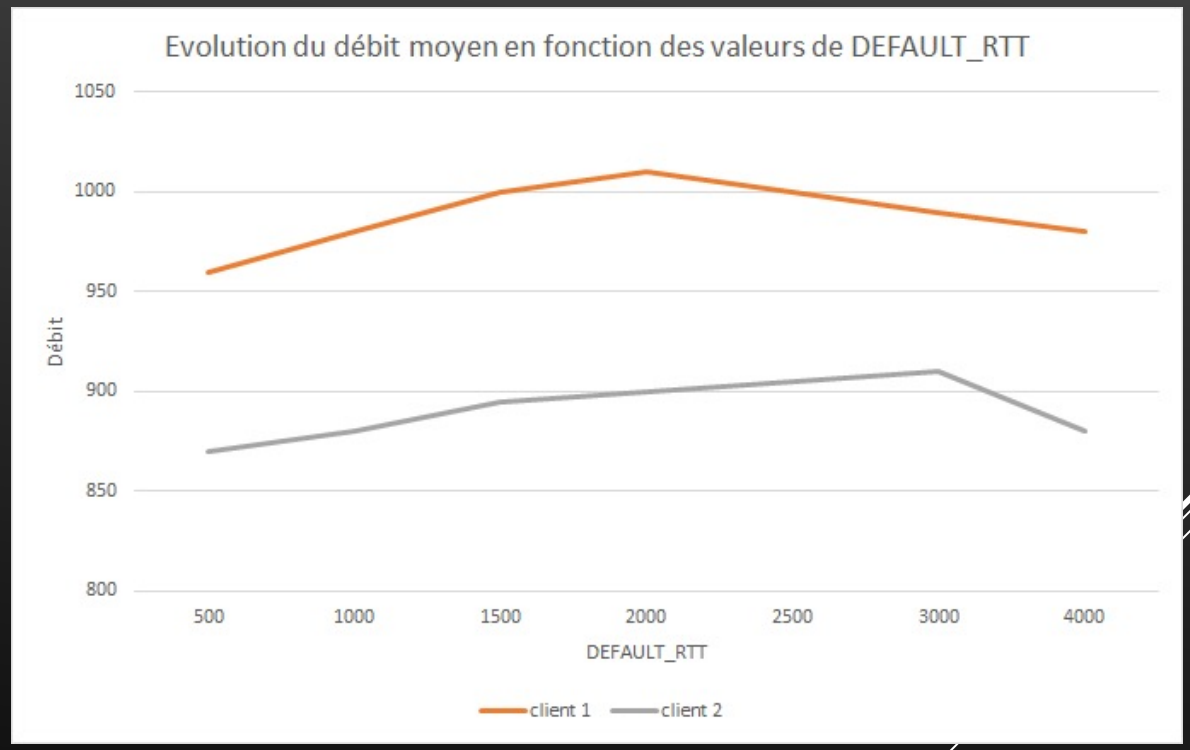
On cherche à optimiser nos variables internes pour les adapter à chaque clients

CLIENT 1

DEFAULT_RTT 2000 µs
ALPHA 0.01
BETA 0.05

CLIENT 2

DEFAULT_RTT 3000 µs
ALPHA 0.01
BETA 0.05



RECHERCHE D'OPTIMISATION DES FACTEURS ET CONSTANTES INTERNES

On cherche à optimiser nos variables internes pour les adapter à chaque clients

CLIENT 1

sRTT : 750 µs
devRTT : 100 µs
Alpha : 0.01
Beta : 0.05

CLIENT 2

sRTT : 1000 µs
DevRTT : 100 µs
Alpha : 0.1
Beta : 0.05

4. RÉSULTATS ET CONCLUSIONS

Voici nos débits moyens pour chaque scénario

Scénario 1 : 1 MB/s

Scénario 2 : 900 KB/s

Scénario 3 : variable en fonction du nombre de client
(alentours de 700kB/s à 800kB/s)

Le client 2 à bien plus tendance à perdre des paquets que le premier (d'où la différence de débit)

Le scénario 3 à des résultats qui dépendent du nombre de clients connectés

Les résultats varient énormément (±100kB/s) en fonction de l'environnement d'exécution

5. PROBLÈMES RENCONTRÉS

- Enormément d'envois : paquet 3000 est le 18000ème envoi.

  Car beaucoup de timeout

  (Peut être lié au code du client)

6. AMÉLIORATIONS FUTURES

- Eviter au mieux les timeouts
- Plus de régularité dans le débit
- Multi-process : envoyer et écouter en même temps ?