PROJET RESEAUX & SYSTEMES

SERVEUR : ZLABYA

ETUDIANTS : IGHILAZA ELYES , HSAINE BADR

ENSEIGNANTS : RAZVAN STANICA, FRANÇOIS LESUEUR

DESCRIPTION DES SCÉNARIOS

SCÉNARIO 1

Un seul client de type 1 à la fois

Pas de congestion au niveau du réseau

CHALLENGE :

Profiter de l'absence de congestion pour un avoir une grande valeur du cwnd mais sans submerger le client ne sachant pas son rwnd

SCÉNARIO 2

Un seul client de type 2 à la fois

Simulation de congestion au niveau du réseau

CHALLENGE :

Bien estimer le RTT pour détecter la perte de paquets

Gérer le Fast Retransmit et le Fast Recovery

SCÉNARIO 3

Quatre clients de type 1 simultanément

Pas de congestion au niveau du réseau

CHALLENGE :

Gestion parallèle des envoies

ZLABYA
Les mécanismes communs

CRÉATION SOCKET

```c
int init_socket(struct sockaddr_in my_addr, socklen_t len, int reuse, int port){
    /*Socket UDP Creation*/
    int sockfd;
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0){
        perror("ERROR opening socket UDP");
        exit(-1);
    }
    /* Allow to reuse the socket */
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
    /* Initialize server address structure */
    memset(&my_addr, 0, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_addr.s_addr = INADDR_ANY;
    my_addr.sin_port = htons(port);
    /*Bind de la structure UDP*/
    if (bind(sockfd, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0){
        perror("ERROR on binding UDP");
        exit(-1);
    }
    /* return socket description */
    return sockfd;
}
```

INITIALISATION DE LA CONNEXION

```c
int init_connect(int sockfd, struct sockaddr_in client_addr, socklen_t len, int port){
    char buf[SIZE_BUF];
    char *ack = "ACK";
    char *syn = "SYN";
    char syn_ack [12];
    strcpy(syn_ack,"SYN-ACK");
    char new_port[5];
        /* clean mem of client adress*/
    memset(&client_addr,0,sizeof(client_addr));
        /* We wait for a SYN */
    memset(buf,0,sizeof(buf));
    recvfrom(sockfd, buf,sizeof(buf), 0, (struct sockaddr *) &client_addr,&len);
        /* check if what we received is a SYN */
    if(strcmp(buf,syn) == 0){
        printf("syn recu \n");
            /* if we did, we send SYN-ACKNewPort */
        sprintf(new_port,"%d",port);
        strcat(syn_ack,new_port);
        sendto(sockfd, syn_ack, sizeof(syn_ack), 0, (struct sockaddr *) &client_addr,len);
        printf("I sent %s\n",syn_ack);
            /* We wait for an ACK */
        memset(buf,0,sizeof(buf));
        recvfrom(sockfd, buf,sizeof(buf), 0, (struct sockaddr *) &client_addr,&len);
            /* We check if what we received is an ACK */
        if(strcmp(buf,ack) == 0){
            printf("ack recu\n");
            return 1;
```

ACQUITTEMENT SUR LE PORT DATA

Plus simple lorsqu'on doit gérer plusieurs clients avec un processus dédié à chaque client.

UN THREAD POUR LA GESTION DES ACK

```c
/* Start the thread to handle ACKs */
pthread_t check_ACK_thread;
struct check_ACK_args context;
context.len = len;
context.sockdata = sockdata;
context.client_addr = client_addr;
context.plast_seg_acked = &last_seg_acked;
context.pwindow = &window;
context.pwindow_sem = &window_sem;
context.pretransmit = &retransmit;
context.pfast_recovery = &fast_recovery;
context.total_nb_seg = total_nb_seg;
context.rtt_table = rtt_table;
if(pthread_create(&check_ACK_thread, NULL, &check_ACK, (void *)&context) != 0){
    printf("error creating check_ACK thread \n");
    exit(-1);
}
```

BOUCLE D'ENVOIE DES SEGMENTS

```
    /* Send the file */
    while(last_seg_acked != total_nb_seg){
        if(retransmit == 1){
            retransmit = 0;
            if(fast_recovery == 1){
                fast_recovery = 0;
                sem_wait(&window_sem);
                window = last_seg_sent - last_seg_acked + CWND_MIN;
                sem_post(&window_sem);
            }
            last_seg_sent = sendData(fd, sockdata, client_addr, len, &window, last_seg_acked, &window_sem, rtt_table);
        }
        else if(last_seg_sent != total_nb_seg){
            last_seg_sent = sendData(fd, sockdata, client_addr, len, &window, last_seg_sent, &window_sem, rtt_table);
        }
    }
    /* after sending the last seg, we wait for the ACK and then we send FIN */
    char fin[4] = "FIN";
    sendto(sockdata, fin, sizeof(fin), 0, (struct sockaddr *) &client_addr,len);
    printf("last_calculated_ack = %d\n and last_seg_acked = %d\n",(file_Size(fd)/SIZE_DATA)+1,last_seg_acked);
    /* We close the file and terminate the process */
    fclose(fd);
    exit(1);
}
```

UTILISATION DE FSEEK() POUR LA RETRANSMISSION

```
    /* We reach the seg we want to send in the file */
fseek(fd, seg_start*SIZE_DATA, SEEK_SET);
```

SLOW START

```
if(*(args->plast_seg_acked) < ack_nb){
    *(args->plast_seg_acked) = ack_nb;
    sem_wait(args->pwindow_sem);
    (*(args->pwindow)) = (*(args->pwindow))+2;
    sem_post(args->pwindow_sem);
}
```

```
    /* We send segs until the window size equals zero */
while(1){
    sem_wait(pwindow_sem);
    if((*window) > 0){
```

FAST RETRANSMIT & FAST RECOVERY

```
else if(*(args->plast_seg_acked) == ack_nb){
    nb_same_ack++;
    if(nb_same_ack == 4){
        nb_same_ack = 0;
        (*(args->pretransmit)) = 1;
        (*(args->pfast_recovery)) = 1;
    }
}
```

```
if(retransmit == 1){
    retransmit = 0;
    if(fast_recovery == 1){
        fast_recovery = 0;
        sem_wait(&window_sem);
        window = last_seg_sent - last_seg_acked + CWND_MIN;
        sem_post(&window_sem);
    }
    last_seg_sent = sendData(fd, sockdata, client_addr, len,
}
```

ESTIMATION DU RTT

```
int estimateSRTT(struct timespec rtt_table[], int seg_number, int sending, int receiving){
    if(sending == 1 && receiving == 0){
        clock_gettime(CLOCK_REALTIME,&(rtt_table[seg_number]));
        return 1;
    }
    else if(sending == 0 && receiving == 1){
        struct timespec tmp;
        clock_gettime(CLOCK_REALTIME,&tmp);
        rtt_table[seg_number].tv_nsec = (tmp.tv_nsec - rtt_table[seg_number].tv_nsec);
        tt_table[0].tv_nsec = 0.6*(rtt_table[0].tv_nsec) + 0.4*(rtt_table[seg_number].tv_nsec);
        return 1;
    }
    else{
        printf("error! estimateSRTT;");
        return -1;
    }
}
```

GESTION DU TIME OUT

```c
void *timer_fc(void *context){
    struct timer_args *args = (struct timer_args *) context;
    struct timespec start, finish;
    while(1){
        long time_out;
        clock_gettime(CLOCK_REALTIME, &start);
        time_out = start.tv_nsec + (args->rtt_table[0].tv_nsec);
        clock_gettime(CLOCK_REALTIME, &finish);
        while(finish.tv_nsec < time_out){
            clock_gettime(CLOCK_REALTIME, &finish);
        }
        sem_wait(args->pwindow_sem);
        *(args->pwindow) = CWND_MIN;
        sem_post(args->pwindow_sem);
        *(args->pretransmit) = 1;
        usleep(500000);
    }
}
```

Zlabya 3

FORK() POUR LA GESTION PARALLÈLE

```c
/* Creat socket: Server_control - Client*/
int sockfd;
struct sockaddr_in my_addr;
struct sockaddr_in client_addr;
sockfd = init_socket(my_addr, len, REUSE, port);
/* Now wait for clients to serve */
while(1){

    port++;
    int pid = fork();

    /* son */
    if(pid == 0){

        /* init un mutex sur le cwnd */
        sem_t window_sem;
        if(sem_init(&window_sem, 0, 1) ==-1){
            printf("error creating window semaphore\n");
            exit(-1);
        }

        /* Creat socket: Server_data - Client*/
        int sockdata;
        struct sockaddr_in data_addr;
        sockdata = init_socket(data_addr, len, REUSE, port);

        /*Wait for the file name */
        printf("waiting for the file name\n");
        recvfrom(sockdata, file_name, sizeof(file_name), 0, (struct sockaddr *)
        file_name[SIZE_DATA-1]='\0';
        printf("got this file name : %s\n",file_name);
```

AMÉLIORATION

UN TIMER INDÉPENDANT SUR CHAQUE SEGMENT

GESTION DU CONGESTION AVOIDANCE

UNE RETRANSMISSION PLUS EFFICACE EN GÉRANT MIEUX LA FENÊTRE DE CONGESTION