Informe Tarea 3

INFO239 – Comunicaciones

Franco Bocca

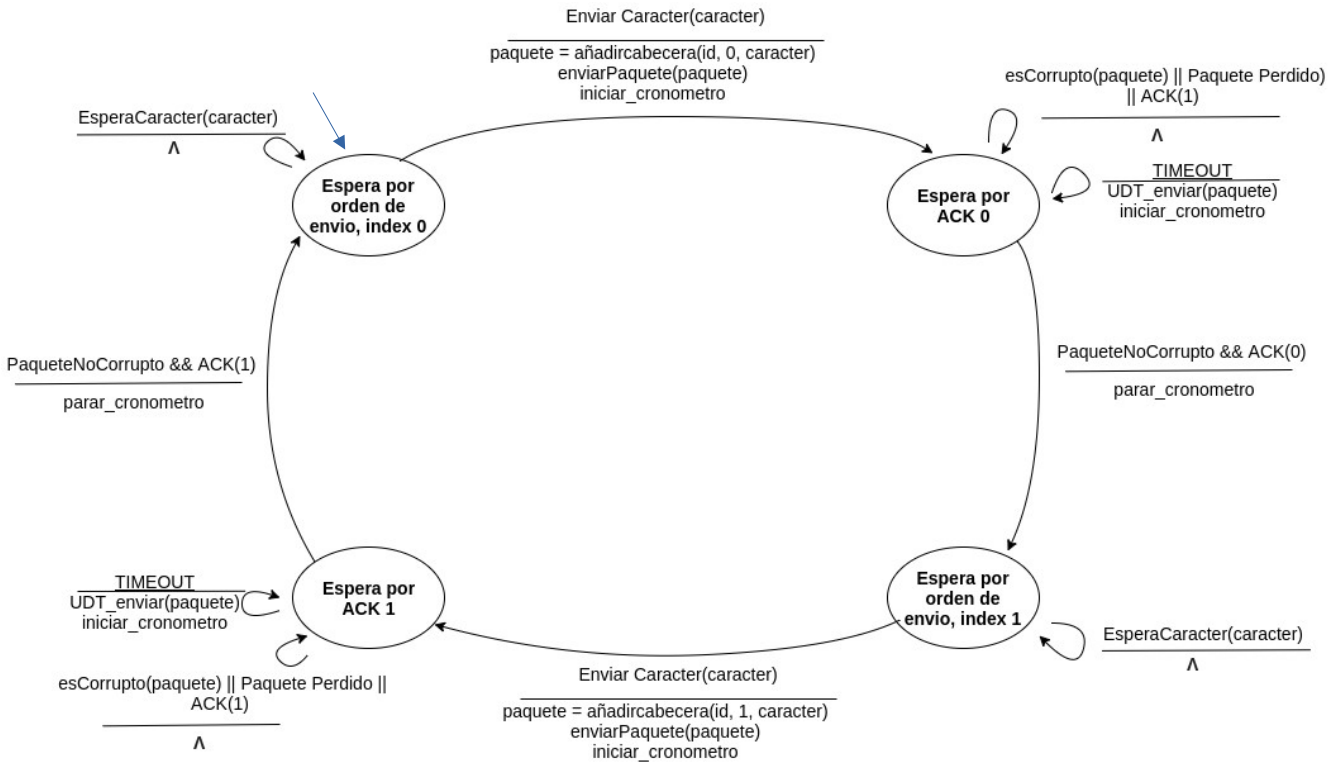
En la realización de esta tarea se decidió partir por el código entregado por el profesor. En primera instancia se trabajo con el código para el servidor, se implemento la perdida aleatoria del 30% y el retardo entre 500 y 3000 ms, además de ir indicando en el bash los paquetes que se perdían. En general lo que fue el desarrollo del servidor no fue tan complejo como esperaba. Una vez ya teniendo un servidor funcionando con lo mínimo que se pedía se prosiguió con el cliente, cuyo desarrollo si se hizo mas complicado. De partida no sabía realmente como hacer un timeout, tuve que tirar mano de la documentación de UDP, hasta que dí con el uso del "settimeout", antes de eso intenté hacerlo de manera "casera", pero hacerlo hubiera sido muy engorroso, porque significaba tener un timer, corriendo de manera paralela mientras el socket esta en modo escucha esperando la respuesta del servidor, así que se decidió por simplemente setear el timeout del socket del cliente en los 2 segundos indicados. Subsanado este primer obstáculo, el siguiente obstáculo fue el manejo de las excepciones. Se hizo mediante un try/except simple dentro de un for que va recorriendo el nombre a enviar, pero cada vez tuve que ir colocando mas loops, para repetir los intentos de reenvío del paquete, al principio era un solo while, después tuve que añadir otro cuando se implementaron los bits 0 y 1 para esperar el ack correcto, Eso me tuvo trabajando reescribiendo parte del código del servidor, porque me obligó a implementar un buffer, el cual no tenia en consideración en un principio, para implementar un buffer, use la estructura de diccionarios los encuentro bastante útiles porque permiten bastante flexibilidad, así cada id del cliente hace de "key", cabe destacar que en un principio usaba los "address" que asigna udp al cliente como id's, pero al final se decidió usar números naturales desde el 1 hasta los n clientes, y se añadió como cabecera al paquete a enviar adicional al bit de control 0 y 1, así pudiendo mantener un control más fácil sobre lo que estaba haciendo, con los bits de control 0 y 1 y los id's listos pude identificar cuando un paquete venia duplicado. Si hay un duplicado en el servidor tal como está establecido en mi FSM, se envía una confirmación de envío, independiente si el paquete esta duplicado o no, ya que se asume que la confirmación de envío pudo haberse extraviado, después de implementar esto, el servidor quedó listo definitivamente, posteriormente apliqué un par de optimizaciones en el código del cliente y servidor, aparte de ir comentando lo que hace el código y la realización de numerosos tests, cosa que se fue haciendo desde el minuto 1 (el comentado y los tests). Con todo lo anterior listo se procedió a probar el sistema con múltiples clientes, donde todo parecía funcionar bien en un principio, hasta que eventualmente el canal se terminaba saturando y los clientes y el servidor terminaban atrapados en un loop infinito de mensajes duplicados a la espera de recibir la confirmación correcta, para solucionar esto fui modificando paulatinamente el código del cliente y servidor hasta que llegó un punto donde simplemente no encontraba forma de solucionarlo, así que deshice los cambios que había hecho, manteniendo solo algunas optimizaciones adicionales y decidí matar dos pájaros de

un tiro, implementando threading para tener múltiples clientes en un solo .py y sincronizándolos mediante la estructura de control más simple de todas; “un semáforo”, de esta forma declaré a la interacción entre un cliente y el servidor como región crítica. Después de un par de pruebas pude ver que efectivamente el problema de saturación se corrigió y ya no tuve más loops infinitos. Lo último que se hizo fue añadir un par de métricas de rendimiento tanto en los clientes como en servidor y colores para los prints, para poder interpretar la información de control más fácilmente.

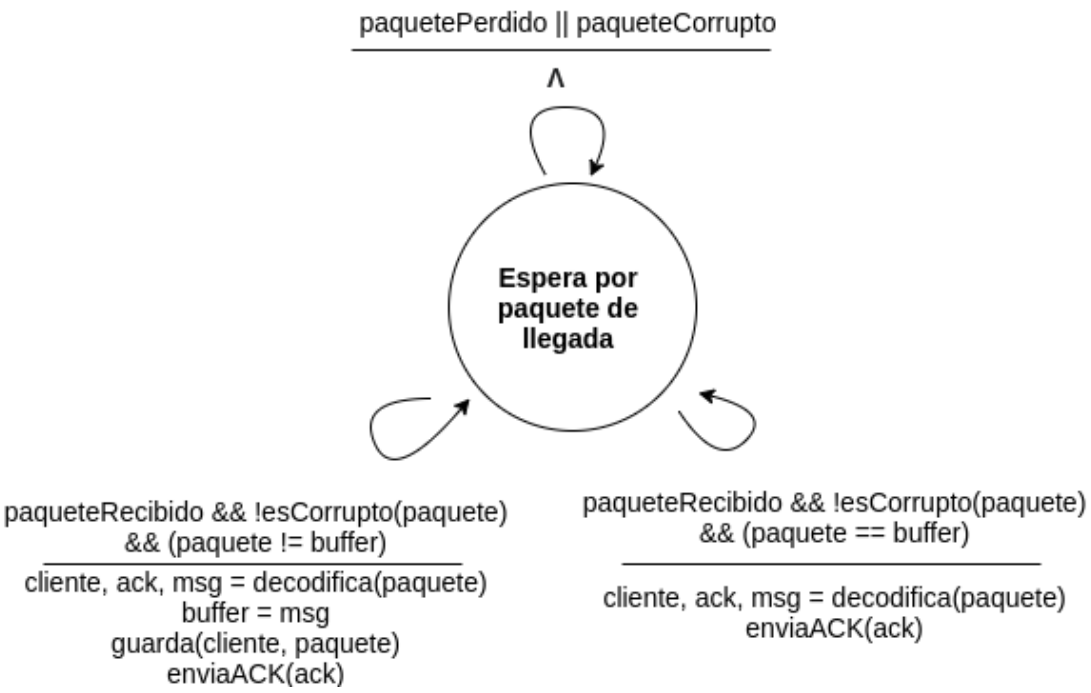
Una vez implementadas las métricas de rendimiento me di cuenta de que en general el rendimiento de este canal de comunicación es pésimo, con 5 clientes al menos 3 presentaron una tasa de pérdidas superior al 60%, demorando mas de 100 segundos en poder enviar el mensaje carácter por carácter, y si el número de clientes aumenta, estas tasas igual lo hacen. En gran parte de esto influye los 3000ms de retardo máximo en el servidor, y que la tasa de pérdida obligatoria es del 30%

Los FSM usados que guiaron el código fueron estos.

EMISOR



RECEPTOR


paquetePerdido || paquetePerdido
Λ
Espera por paquete de llegada
paqueteRecibido && !esCorrupto(paquete) && (paquete != buffer)
cliente, ack, msg = decodifica(paquete)
buffer = msg
guarda(cliente, paquete)
enviaACK(ack)
paqueteRecibido && !esCorrupto(paquete) && (paquete == buffer)
cliente, ack, msg = decodifica(paquete)
enviaACK(ack)