Proyecto: Eliminating Distributed Receive Livelock

Dalia Camacho, Gabriela Vargas

I. Introducción

Como se menciona en el artículo, la mayoría de los sistemas operativos utilizan interrupciones como método para planificar las tareas relacionadas con eventos I/O [1]. Estas interrupciones tendrán una prioridad más alta para el planificador. Sin embargo, se puede llegar al punto de que el planificador se pase todo el tiempo atendiendo interrupciones de los I/O con el riesgo de afectar la ejecución de tareas vitales.

Aunque existe el riesgo anteriormente mencionado, las interrupciones son una alternativa viable cuando su tasa de llegada es baja, ya que generan un overhead bajo, pero su viabilidad se ve comprometida cuando la tasa de llegada es alta. A esta sobrecarga se le conoce como *livelock*.

Ante esta situación, los autores del artículo presentan un enfoque de combinación de estrategias de planificación alternando entre técnicas de poleo e interrupciones cuando la tasa de llegada de llegada de paquetes es alta. Al presentarse un *livelock*, el planificador cambia de estrategia de interrupciones a una estrategia de poleo basado en un enfoque *Round Robin*, el cual tiene implicaciones de incremento de latencia en cuanto a la respuesta de los eventos, por lo que sería contraproducente implementarlo en un escenario donde las interrupciones son escasas.

Entonces, la propuesta de los autores se basa en garantizar rendimiento y mejorar las latencias bajo sobrecarga mientras que se mantienen las ventajas de un sistema de interrupciones cuando hay tasas bajas de llegadas de eventos.

En este trabajo, simulamos el sistema propuesto por los autores del artículo y ejecutamos una serie de experimentos midiendo su rendimiento en cuanto a las métricas de *throughput* y latencia para determinar si esta propuesta efectivamente evita la condición de *livelock* y genera las mejoras mencionadas por los autores.

II. Descripción de la arquitectura del sistema

El sistema engloba los dispositivos de entrada; el kernel donde se encuentra el administrador de paquetes de llegada el cual tiene implementadas las tres estrategias posibles que son poleo, manejador de interrupciones y una combinación entre ambas; y las aplicaciones que son las que reciben los paquetes enviados por los dispositivos de entrada.

Los dispositivos de entrada se conectan con el kernel mediante puertos, cada dispositivo de entrada tiene un puerto único. A su vez, el kernel se conecta de forma similar con las aplicaciones donde cada aplicación tiene un puerto que le corresponde. El sistema fue creado de tal forma que el dispositivo uno se conecta con la aplicación uno, el dos con la aplicación dos y así sucesivamente. El número de aplicaciones desplegadas puede ser definido externamente por el usuario.

Program	Argument	Description
Device	fun	Función de envío de paquetes
	wait	Parámetro de función constante
	lambda	Parámetro de función Poisson
	mu	Parámetro de función normal
	var	Parámetro de función normal
	inc	Parámetro de función picos
	dev	Número de dispositivos
	print	Impresión en pantalla los envíos y resultados
	maxtime	Tiempo máximo de envío de paquetes en minutos
	lag	Tiempo añadido al final para que los otros terminen
	eval_lat	Evaluación de latencia
	experiment	Nombre del experimento con el que se guarda el archivo

Table I
ARGUMENTOS QUE EL USUARIO PUEDE DAR AL PROGRAMA `DEVICE.PY`

Los dispositivos de entrada pueden tener distintas tasas de envío controladas por funciones parametrizadas. Las aplicaciones únicamente se encuentran a la espera de paquetes. Lo que ocurre dentro del kernel para el manejo de paquetes es más complicado. En el caso de estrategia combinada o de interrupciones, éste cuenta con un buffer en el que todos los paquetes se van acumulando en espera de ser atendidos y enviados a la aplicación correspondiente. Por otro lado está un manejador de interrupciones que lee el contenido del buffer y se encarga de enviar los paquetes a las aplicaciones. En el caso de la estrategia combinada se cambia de manejo de interrupciones a poleo cuando el buffer se llena y regresa nuevamente a manejo de interrupciones cuando hay al menos cierto espacio disponible en el buffer. Para la estrategia de poleo se monitorean todos los puertos bajo una estrategia der *round robin*.

III. Información para el despliegue del programa

La implementación se hizo principalmente en tres programas de python `device.py`, `kernel.py` y `apps.py` cada uno tiene parámetros que pueden ser modificados por el usuario los cuales se encuentran en las Tablas I a III.

Además de estos tres programas se definió el código `tiempo.py` en el cual se encuentran todas las funciones que fueron implementadas para generar distintas tasas de envío de paquetes.

La versión de Python que se utilizó fue la 3.5.6 para todos los códigos hasta ahora mencionados. Para poder ejecutar los códigos encontramos dos formas posibles para la primera se necesita lo siguiente:

1) Abrir tres terminales
2) En cada terminal definir el directorio de trabajo
3) En la primer terminal correr `python apps.py`
4) En la segunda terminal correr `python kernel.py`
5) En la tercer terminal correr `python device.py`

Program	Argument	Description
Kernel	calendar	Estrategia de manejo de recepción paquetes
	host	Parámetro para comunicación UDP
	port1	Parámetro para comunicación UDP con dispositivos
	port2	Parámetro para comunicación UDP con aplicaciones
	sz	Tamaño del buffer
	perc_occup	Porcentaje de ocupación máximo del buffer para regresar a manejo de interrupciones
	print	Impresión de recepciones, envíos y resultados
	lag	Tiempo añadido al final para que los otros terminen
	experiment	Nombre del experimento con el que se guarda el archivo

Table II
ARGUMENTOS QUE EL USUARIO PUEDE DAR AL PROGRAMA `KERNEL.PY`

Program	Argument	Description
Apps	PORT	Parámetro para comunicación UDP con aplicaciones
	print	Impresión de recepciones, envíos y resultados
	lag	Tiempo añadido al final para que los otros terminen
	experiment	Nombre del experimento con el que se guarda el archivo

Table III
ARGUMENTOS QUE EL USUARIO PUEDE DAR AL PROGRAMA `APPS.PY`

Es necesario seguir el orden antes mencionado ya que el programa que replica las aplicaciones está a la espera de envíos por parte del kernel, si los envíos comienzan antes que las aplicaciones estén listas la información se pierde. Lo mismo ocurre entre kernel y dispositivos de entrada.

La segunda forma es definiendo un código de bash que contenga al menos lo siguiente:

```
#!/bin/bash
python apps.py &
python kernel.py &
python device.py
```

Para generar los resultados corrimos un código en bash para evitar el problema de las tres terminales y poder generar resultados de forma sistemática.

El equipo en que se ejecutó tiene un procesador AMD A10-5745m apu with radeon(tm) hd graphics $\times$ 4, con 10.9 GiB con sistema operativo Ubuntu 18.04.3 LTS de 64 bits.

IV. Diagramas de secuencia

A. Diagrama de alto nivel

La Figura 6 representa el flujo de comunicación entre cada hilo i que representa un dispositivo que envía paquetes de información que se añaden al buffer implementado en el kernel, en caso de que éste no se encuentre lleno. De acuerdo a un porcentaje de ocupación del buffer, el planificador decidirá la estrategia a seguir según un esquema de interrupciones o un esquema combinado con poleo y posteriormente se enviará el mensaje a la aplicación con hilo j.

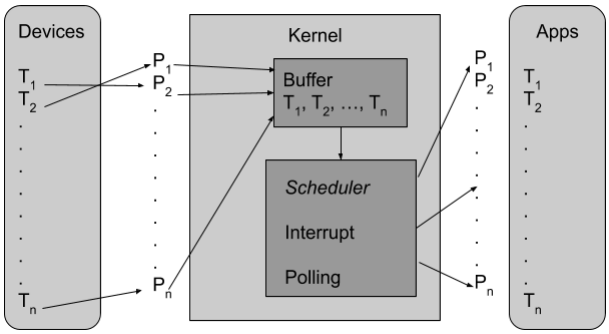

Figure 1. Diagrama de secuencia a alto nivel

B. Diagrama de secuencia UML

En la figura 6 se muestra el flujo de envío de paquetes entre los dispositivos, el kernel y las aplicaciones. En el caso de este experimento es fácil ver que la comunicación es unilateral, ya que la función de las aplicaciones solamente se centrará en la recepción de paquetes.

V. Estrategia Utilizada para implementar el proyecto

Para replicar el comportamiento de las distintas estrategias de manejo de datos de entrada se utilizó comunicación UDP y programación en hilos en Python 3.5.6.

Se crearon tres programas, el primero replica el comportamiento de los dispositivos de entradas y salidas (`device.py`). El segundo simula los procesos que ocurren en el kernel ante la recepción de mensajes ya sea con interrupciones, poleo o la combinación de ambos (`kernel.py`). A su vez, el segundo programa envía los paquetes ya administrados al tercer programa el cual correspondería a las distintas aplicaciones o clientes (`apps.py`).

Para el envío de paquetes del primer programa al segundo se definieron distintas funciones para el envío de paquetes, las cuales pueden ser parametrizadas por el usuario. Dentro de estas funciones se incluyen una cuyo tiempo de espera entre paquetes es constante, otras dos en que el tiempo de espera es aleatorio, el cual puede ser generado muestreando de las distribuciones normal y Poisson. Existe otra función en que el envío pasa de ser nulo a envío constante de forma cíclica, finalmente, se tiene una función en que el tiempo de envío va incrementando hasta un límite superior y después decrece hasta uno inferior. Estas funciones se encuentran en otro código y son accesibles al primero.

El envío de paquetes se realiza por hilos distintos, cada hilo representa un dispositivo de entrada y salida, el número de dispositivos también puede ser definido por el usuario. Cada hilo tiene un puerto asociado al cual envía los paquetes. Además se tiene un puerto general que no está asociado a los hilos cuya función es avisar a los otros programas el número de dispositivos que se tienen y el momento en cual se debe finalizar la ejecución. El tiempo total de envío de paquetes antes de finalizar la ejecución también puede ser definida por el usuario.

El segundo programa es el que administra la llegada de paquetes de los dispositivos de entrada y posteriormente los envía a las aplicaciones. Para la estrategia de interrupciones se definió un buffer en el que los paquetes se iban acumulando. El buffer se definió mediante una cola de tamaño fijo (definido por el usuario), para poder agregar elementos al buffer, se definieron tantos hilos como dispositivos de entrada, donde cada hilo lee un puerto de entrada distinto y manda los paquetes al buffer en cuanto aparecen. Más adelante sólo se utiliza un hilo para entregar los paquetes a las distintas aplicaciones, para saber a qué aplicación corresponde lo que hicimos fue en cada mensaje enviado indicar el número de dispositivo y el número de mensaje. A partir de esto sólo se tiene que leer el mensaje e identificar el número de dispositivo. Esto último lo hace el manejador de interrupciones que lee el primer elemento de la cola y lo envía.

En caso que se tenga la estrategia mixta todo lo anterior se mantiene, la única diferencia es que si el buffer está lleno, entonces se cambia de estrategia a poleo hasta que al menos un $x\%$ del buffer esté libre. La estrategia de poleo no requiere del uso del buffer, simplemente un hilo va pasando por cada puerto y si encuentra un mensaje, lo envía. En el caso de poleo la estrategia combinada la política *round robin* aparte de contempla al buffer además de los puertos; de esta manera el buffer puede irse liberando poco a poco.

Por último el segundo programa cuenta con un hilo que está esperando la llegada de un mensaje del primer programa el cual indica que se debe terminar la ejecución. Una vez que llega, este mensaje se envía al programa que replica la aplicaciones.

El tercer programa correspondiente a las aplicaciones primero se encuentra a la espera de conocer el número de dispositivos que se están ejecutando para poder desplegar el mismo número de aplicaciones y definir los puertos correspondientes. Cada hilo recibe los paquetes que se le envían a través del segundo programa. Un hilo adicional escucha un puerto general a la espera del mensaje de detención, ante el cual la ejecución se detiene.

En los tres programas se tienen variables respecto a las métricas que se pueden evaluar, como número de paquetes enviados, tiempo al momento de envío, número de paquetes recibidos, tiempo al momento de llegada, número de veces en que se llenó el buffer y número de cambios de estrategia. La cantidad de mensajes se utilizó para medir throughput y el tiempo de envío y recepción para medir la latencia. Resultados de esto se muestra más adelante.

VI. Experimentos

Los experimentos que se realizaron tuvieron la finalidad de evaluar las tres estrategias de recepción de paquetes y se obtuvieron el *throughput*, la latencia, el número de veces que se llenó el buffer en cada caso y el número de paquetes que se perdieron.

Todos los experimentos se hicieron en unequipo con procesador AMD A10-5745m apu with radeon(tm) hd graphics $\times$ 4, con 10.9 GiB con sistema operativo Ubuntu 18.04.3 LTS de 64 bits.

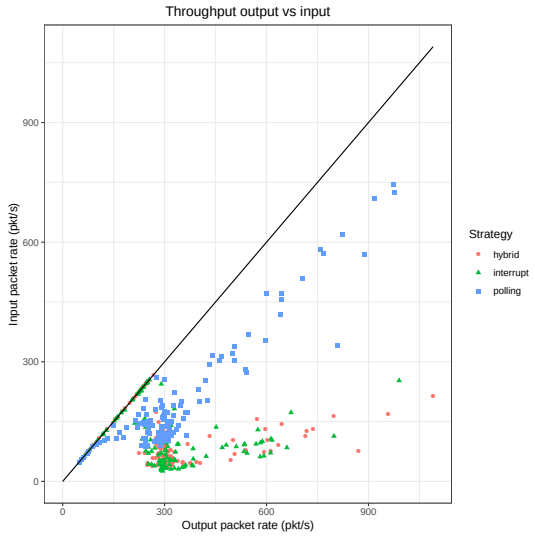

Figure 2. Se muestra el *throughput* de salida y entrada en términos de paquetes/segundo.

Para el experimento se consideraron tres dispositivos de entrada, un buffer de tamaño 100, un tiempo de envío de paquetes de 12 segundos en los cuales los paquetes fueron enviados con un mismo tiempo de espera entre ellos en cada ejecución. Para la estrategia combinada se definió que para cambiar de poleo a manejo de interrupciones era necesario que el buffer estuviera a lo más al 80% de ocupación. Los tiempos de espera variaron de $8.85e-11$ hasta 0.005 segundos cada tiempo de espera siendo 1.176 veces mayor que el anterior.

En la Figura 2 se muestran los paquetes enviados por segundo contra los paquetes recibidos por segundo. Podemos notar que cuando el número de paquetes enviados por segundo es menor a 250 paquetes por segundo las estrategias de manejo de interrupciones y la estrategia combinada logran mantener el mismo *throughput* de recepción y de envío incluso mejor que la estrategia de poleo la cual comienza a perder desempeño alrededor de los 150 paquetes por segundo. Sin embargo alrededor de los 300 paquetes por segundo el desempeño de las primeras dos estrategias se ve mermado, la razón más factible es por que bajo esa tasa de envío ocurre un *live lock* del cual no se puede recuperar.

A pesar de lo que hubiéramos esperado el desempeño no mejora con la estrategia combinada, si no que se ve muy parecido al de manejo de interrupciones. Por otro lado podemos ver un comportamiento interesante del poleo ya que al igual que las otras dos estrategias el *throughput* decrece alrededor de los 300 paquetes por segundo. Sin embargo vuelve a despuntar logrando un throughput de recepción de paquetes de hasta 745 paquetes por segundo. Esto podría deberse a que no tiene que esperar a la recepción de paquetes en ninguno de los puertos, por que estos se encuentran listos para recibirse.

En la Figura 3 se muestra el número de paquetes perdidos contra los paquetes enviados por segundo. De igual manera podemos observar que en un inicio la estrategia de manejo de interrupciones y la estrategia híbrida casi no pierden paquetes, pero con poleo se empiezan a perder paquetes muy pronto. Sin embargo alrededor de los 300 paquetes por segundo enviados,

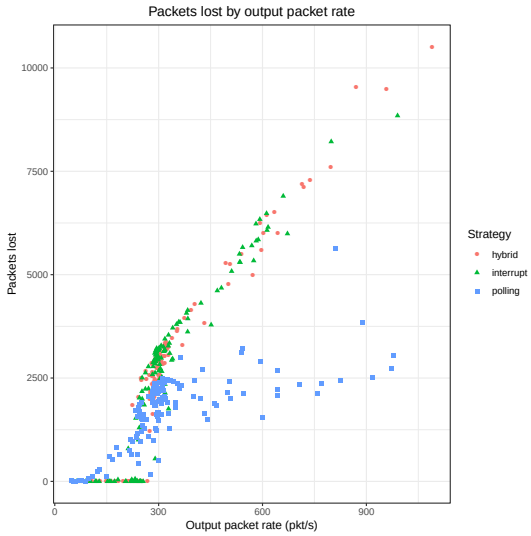

Figure 3. Número de paquetes perdidos en términos de paquetes enviados por segundo.

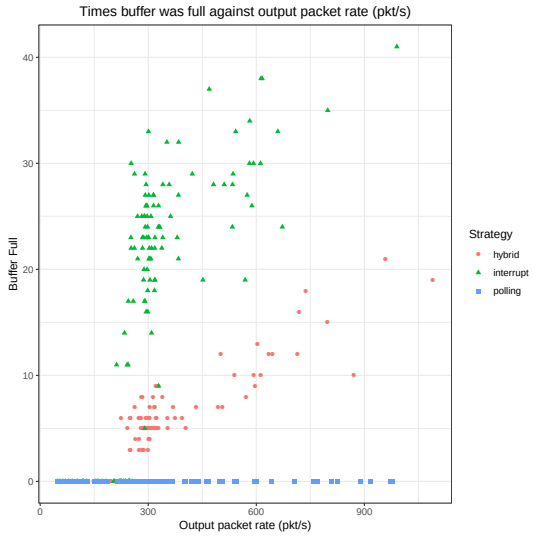

Figure 4. Número de veces en que se llenó el buffer en términos de paquetes enviados por segundo.

los paquetes perdidos crecen con un salto que supera los paquetes perdidos por poleo. Después de eso la pérdida de paquetes sigue incrementando en los primeros dos casos pero se estabiliza bajo la estrategia de poleo.

La Figura 4 muestra el número de veces que se llenó el buffer bajo cada estrategia, sabemos que en poleo no hay buffer, por lo tanto éste nunca se llena. Para los casos de las estrategias de manejo de interrupciones y la estrategia híbrida vemos lo esperado, que en la estrategia híbrida el buffer no se llene tantas veces como ocurre cuando sólo se tiene manejo de interrupciones. A pesar de que esto es tal como esperábamos, el desempeño en cuanto a *throughput* no mejora bajo la estrategia combinada. Por lo que una posible explicación sería que la implementación es ineficiente ya que se hacen bastantes condicionales y el cambio de estrategia hace que se pierda tiempo. Además, si lo pensamos en términos de ejecución especulativa cada vez que cambie de estrategia va a haber una ejecución fallida debido a una mala predicción por lo es necesario regresar al paso anterior. Esto no ocurre en poleo ya que los condicionales siempre indican que la estrategia a mantener debe ser la estrategia de poleo. Tal vez incrementar el espacio que debe estar libre en el buffer antes de regresar a manejo de interrupciones podría ayudar a disminuir el número de cambios de estrategia. También se podría evaluar tamaños de buffer mayores.

No se realizó una evaluación adicional sobre los cambios de estrategia, ya que estos son dos veces el número de veces que se llena el buffer en la estrategia combinada. Ya que cuando se llena inicia el poleo y si se vuelve a llenar quiere decir que pasó de poleo a menejo de interrupciones y nuevamente a poleo. Por lo que también observamos que el cambio de estrategia incrementa cuando el número de paquetes enviados por segundo incrementa.

Relacionando las Figuras 2 y 4 podemos notar que el desempeño en términos de *throughput* de recepción de paquetes empeora una vez que el buffer se llena.

Finalmente, evaluamos la latencia, es decir el tiempo que

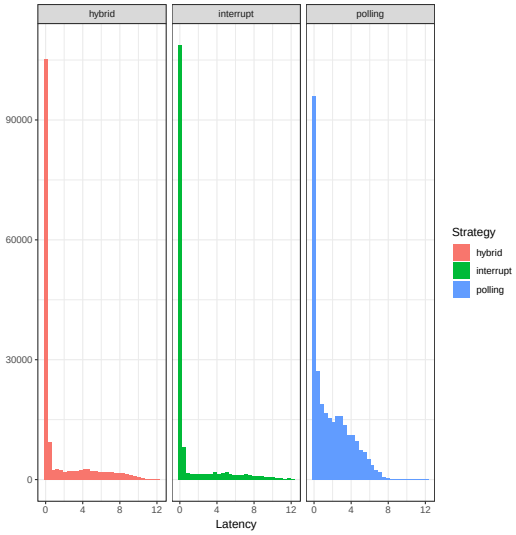

Figure 5. Latencia o tiempo de espera entre envío y llegada de los paquetes que no se perdieron.

tarda en recibirse un paquete después de ser enviado. La medida de latencia sólo la consideramos para los paquetes que sí fueron recibidos. Podemos observar que en las estrategias híbridas y de manejo de interrupciones la distribución de la latencia tiende a ser baja en la mayoría de los casos, pero tiene una cola larga. En cambio en poleo la distribución no está tan concentrada en el cero, pero no tiene una cola tan larga. La diferencia en las distribuciones se puede deber a que lo que entra al buffer se envía en cuanto se puede, pero con poleo el round robin es un poco más lento. Para poder evaluar mejor las desventajas de poleo en cuanto a latencia hubiera sido necesario repetir los experimentos, pero para un número mayor de dispositivos de entrada.

VII. Lista de sugerencias para mejorar los algoritmos

1) Para calcular la latencia a la cual se envía un paquete, implementamos una lista anidada a la que le vamos añadiendo los datos de tiempo de recepción de mensajes y mensaje enviado para el thread i. Esto supone una llamada a la rutina que añade elementos a la lista cada que se recibe un mensaje, por lo que una mejora al algoritmo podría ser implementar otro mecanismo de cálculo de latencia con el fin de que el paso de recolección de datos no tenga un impacto en el desempeño de los experimentos.
2) Los resultados de nuestro experimento dependen de la computadora utilizada y del entorno de programación elegidos. De igual forma, consideramos que el diseño que le dimos al sistema simulado influye en los resultados que obtuvimos, por lo que nuestras conclusiones están restringidas a estos detalles y no podemos decir de forma concluyente que la estrategia de combinación de interrupciones y poleo no tiene mejoras significativas con respecto a un esquema de solo interrupciones.

VIII. Conclusiones

En general pudimos observar que para el manejador de interrupciones sí ocurre el efecto de *live lock* y que en un inicio tiene mejor desempeño que poleo. También pudimos ver que el número de veces que el buffer se llena es menor en la estrategia combinada. A pesar de esto el desempeño en cuanto a *throughput* fue muy similar al de manejo de interrupciones, por lo que al menos en los experimentos que realizamos la estrategia combinada no mejora el desempeño ni evita el *live lock*. Para poder llegar a conclusiones más contundentes sobre la utilidad de la estrategia combinada se tiene la opción de aumentar el tamaño de buffer o disminuir los cambios de estrategia al requerir un mayor espacio libre en el buffer antes de regresar a la estrategia de manejo de interrupciones. Por otro lado sólo se comparó contra poleo con tres dispositivos, habría que ver si el comportamiento de poleo se mantiene con más dispositivos de entrada.

References

[1] Jeffrey C Mogul and KK Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS)*, 15(3):217–252, 1997.

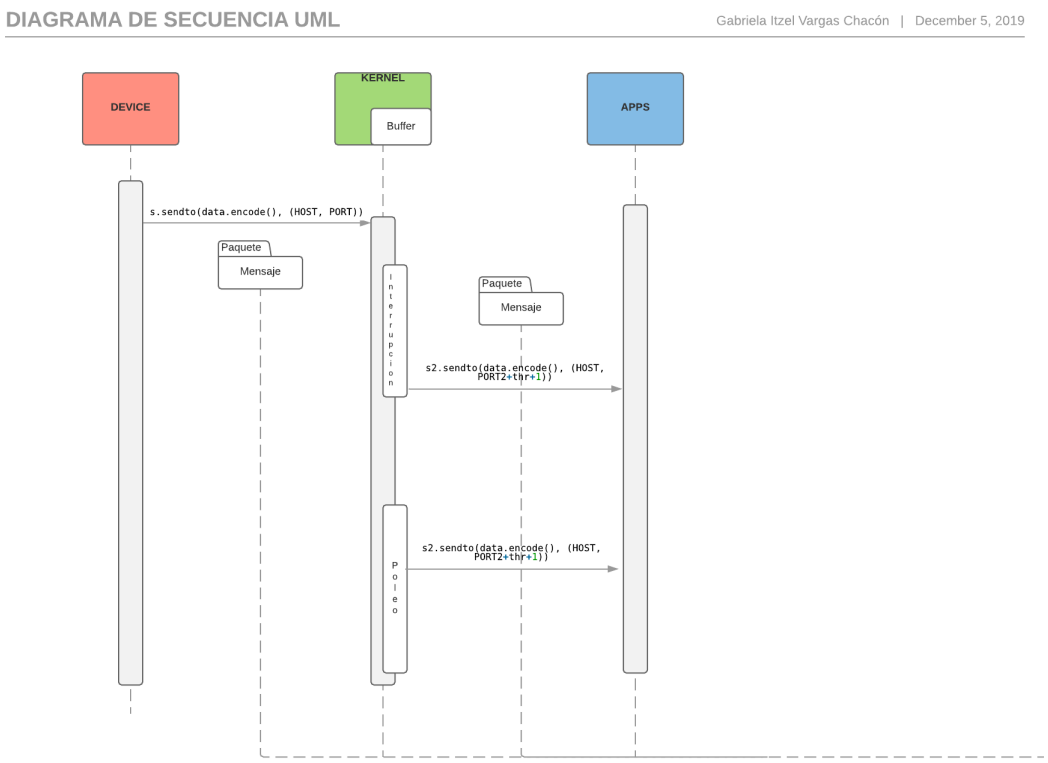

Figure 6. Diagrama de secuencia a alto nivel