



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN
IIC2523 - SISTEMAS DISTRIBUIDOS

Tarea 2

18 de noviembre de 2020

2º semestre 2020 - Profesor Cristián Ruz

Maximiliano Schudeck - 16638530

Totally Ordered Multicast

Reloj Lineal

Para esta parte, decidí utilizar en Golang una función principal `main()` que instanciara varias go functions, de manera que concurrentemente se pudieran ejecutar varias instrucciones que enviaran mensajes para distintos receptores cada una. A su vez, la función `main()` tendría una serie de variables que permitieran funcionar como reloj lineal, como lo puede ser el contador, el cual aumentaría dependiendo de los mensajes recibidos y enviados, la queue, necesaria para la implementación del algoritmo, así como de las instrucciones que leyera a partir del archivo `instructions.txt`.

Al enviar los mensajes, implementé separación por espacios en los strings. El string puede dividirse en distintos elementos gracias a esto. El primer elemento es el código del string, implementado así pues se requiere separar ACKs de Mensajes. Los mensajes son enviados con el código 'MSJ' mientras que los ACK's con 'ACK'. Luego, el siguiente valor contenido en el mensaje corresponde a la ID del mensaje, obtenida del archivo `instructions.txt`. Cada instrucción tiene un ID de mensaje correspondiente a su numero de linea.

```
PROCESO 0 TERMINADO
RELOJ LOGICO: 4
QUEUE FINAL:
''
LISTA MENSAJES ENVIADOS Y RECIBIDOS:
'SENDMSJ 0 1', 'INCREASE 2', 'RCVMSJ 2 3'
```

En esta imagen del output del programa, creé una lista que guardara todos los elementos que en algún momento entraron a la queue, es decir, todas las instrucciones que siguió el proceso. Aquí se puede ver, por ejemplo, para el primer elemento de la lista de enviados y recibidos, como envió el mensaje(`SENDMSJ`) con `ID = 0`, y con `valor = 1`, como incrementó el valor de su clock en 2, y como recibió el mensaje con `ID = 2` y `valor 3`. Estos IDs de los mensajes no tienen que ver con el nodo que los manda o recibe, si no simplemente con la linea correspondiente a la instrucción del archivo `instructions.txt`, para garantizar unicidad en la ID de cada mensaje.

Cada proceso también tiene su propio ID, dependiendo del primer puerto que haya encontrado disponible de los puertos listados en `config.txt`. La primera consola en ejecutarse tendrá el ID 0, y en el caso de los archivos de ejemplo, el puerto 3000, la segunda consola tendrá el ID 1, y el puerto 3001, y así. No está harcodeado, así que aunque la primera linea de `config.txt` tenga el puerto 4242, igual tendrá el ID correspondiente al numero de linea, o sea 0. Debe ejecutarse el programa en distintas terminales un numero de veces igual al número de puertos. Una vez se instancie la ultima terminal, es decir, que lea la última linea del archivo, esta última terminal procederá a mandarle a todas las otras direcciones un mensaje diciendoles que deben empezar el programa, para así de esa manera poder correr concurrentemente varios relojes lineales sincronizadamente.

El tiempo de ejecución del programa entero es determinado por una variable preprogramada llamada `millisecondExecution`, la cual indica la cantidad de milisegundos que le tomará al programa terminar. Yo lo dejé en 10000, es decir, 10 segundos, pero si requiere más tiempo de ejecución para la corrección se puede cambiar fácilmente. El último socket es el encargado de enviarle al resto el mensaje de inicio, y también el del final, por lo que esta variable determina cuanto tiempo de ejecución darle de buffer al resto de los sockets para que todos terminen sus procesos.

Cuando termina el proceso, son impresas en pantalla tanto los mensajes enviados, así como tambien los relojes finales, la queue final, y todos los elementos que fueron añadidos en algún momento a la queue.

Implementación algoritmo

Unas de las funciones más importantes son el `main()`, que se encarga de escuchar el puerto UDP escogido al iniciarse el programa, `send_messages()`, que se encarga de iterar por la lista de instrucciones y de ejecutar distintas funciones dependiendo de la operación que sea necesario realizar, `send_acks_global()`, que se encarga de enviar un mensaje tipo `ACK` a todos los puertos especificados en el archivo `config.txt` cuando le corresponde a un proceso hacer acknowledgement de que ha procesado una instrucción. `send_message_to_addr()` el cual se encarga de enviar 1 mensaje a 1 puerto, y `queue_manager()`, que se encarga de constantemente revisar si entra un elemento a la queue, y si el elemento a la cabeza de esta cumple con las condiciones para pasar a ser ejecutado, es esta función la encargada de realizar los cambios al reloj y eliminar el elemento de la queue.

```
Incrementando contador antes de mandar mensaje: MSJ 3 1
Procediendo a enviar mensaje ID: 3 a destinatario con ID: 4
Esperando ACKs del mensaje enviado con id: 3
Han llegado todos los acks correspondientes a proceso 3
Popeando SENDMSJ 3 1
```

Las instrucciones que debe realizar cada nodo son leídas por la función `send_messages()` que pasa por todas las ordenes, y cuando encuentra una orden que debe ejecutar el proceso en específico, lo añade a la queue. Por ejemplo, cuando hay una instrucción que dice que debe enviar un mensaje (una instrucción `M`), añade la instrucción de enviar el mensaje a la queue con un identificador `SENDMSJ`, luego, cuando la instrucción es leída en la queue por `queue_manager()`, incrementa el contador del proceso, envía el mensaje a todos los destinatarios, y se queda esperando los ACKs correspondientes al ID del mensaje enviado, para así cuando lleguen una cantidad de ACKs igual a la cantidad de procesos a las cuales se les envió el mensaje, salir del loop de espera, y pasar a la siguiente instrucción.

Si lee una función que dice que incremente su valor, también es añadido a la queue, de manera que sea leído en orden, esta vez con el identificador `INCREASE` y el valor a incrementar.

```
RECIBIDO MENSAJE RECVMSJ 2 3
RELOJ DE MENSAJE 2 ES MAYOR QUE RELOJ ACTUAL 0
INCREMENTANDO RELOJ A : 3
INCREMENTANDO RELOJ PUES SE RECIBIÓ MENSAJE ID: 2
NUEVO RELOJ: 4
Popeando RECVMSJ 2 3
```

Si se recibe un mensaje, este es añadido a la queue con el identificador `RECVMSJ`, que indica que se recibió un mensaje. Una vez es procesado por `queue_manager()`, este procede a enviar un ACK global, es decir, todos los procesos reciben este ACK, pero solo hay 1 sender esperando por el ACK con el ID especificado, que es a quien estamos buscando que lo reciba. Luego de mandar el ACK, se procede a cambiar el reloj si es que el valor del mensaje recibido era mayor, y además debido a que se recibió un mensaje.

A medida que las instrucciones son leídas existe un pequeño delay, para asegurar que las instrucciones se lean de manera correcta como aparece en el archivo `output.txt`. Sin embargo, todas las funcionalidades cumplen con la modelación del Totally Ordered Multicast. Cuando se manda un mensaje, se comienza a esperar por sus ACKs, y no deja de esperarlos hasta

que llegan todos, por lo que solo puede pasar a la siguiente operación cuando lleguen todos los ACKs correspondientes al ID del proceso. Una vez que un proceso recibe este mensaje, lo pone a la cola, la cual se va reduciendo de manera FIFO. Cuando le toca ser "popeado,"^a un elemento a la cabeza de la queue, se ejecuta la instrucción. De ser un reconocimiento de que le llegó un mensaje(RECVMSJ), procede a enviar un ACK global con la id del mensaje, y a incrementar su reloj según el valor contenido en el mensaje si es necesario, y luego una vez más. Si es una instrucción de enviar mensaje(SENDMSJ), se incrementa el valor del reloj, se envía el mensaje a las direcciones correspondientes y se procede a esperar sus ACKs. Si es de incremento del propio clock(INCREASE), se incrementa su propio clock.

De esta manera, queda implementado Totally Ordered Multicast con todas las instrucciones especificadas.

Causally Ordered Multicast

Reloj Vectorial

Implementé el código de manera similar que para el reloj lineal. Ya no necesitaba el uso de ciertas funciones como aquellas que enviaban ACK's, pero las funciones que iniciaban y terminaban el programa, analizaban la queue, mandaban mensajes y se encargaban de subir el contador cuando se procesaba una instrucción, fueron refactorizadas para que funcionaran con relojes vectoriales.

Para el reloj vectorial, implementé un array de ints de largo N, donde N es el numero de elementos en config.txt. Luego, a cada proceso le sería asignado un ID de valor entre 0 y N-1, de manera que tendría N ID's distintas, una para cada proceso, y para cada elemento del vector. De esta manera, podría incrementar el valor de un reloj según el ID de un proceso, simplemente llamando al ID correspondiente.

```

PROCESO 4 TERMINADO
RELOJ VECTORIAL: [0 0 0 1 1]
QUEUE FINAL:
''
LISTA MENSAJES ENVIADOS Y RECIBIDOS:
RECVMSJ 3 0 0 0 1 0' 'SENDMSJ 4 0 0 0 1 1

```

Al enviar los mensajes, implementé separación por espacios en los strings. El string puede dividirse en distintos elementos gracias a esto. El primer elemento es la instrucción a realizar, que en este caso solo puede ser procesar el envío SENDMSJ, y la recepción de mensajes RECVMSJ. El segundo elemento, corresponde al ID de la instrucción. Por ejemplo, en la imagen, para el primer elemento de todos los mensajes que recibe el proceso, se puede ver como el proceso 4 recibe el mensaje ID = 3, y con un reloj vectorial [0, 0, 0, 1, 0], y luego envía el mensaje con

ID = 4 y con un reloj vectorial igual a $[0, 0, 0, 1, 1]$, pues incrementó su propio reloj antes de enviarlo. Esta implementación se discute más detalladamente en la sección Implementación Algoritmo.

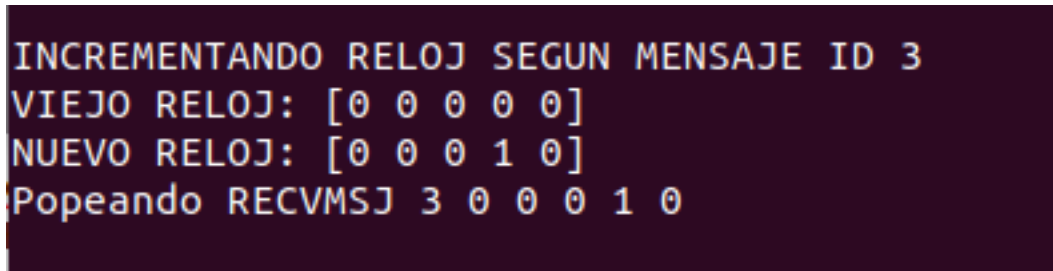
Implementación algoritmo

Para implementar el algoritmo, me basé más que nada en las issues que explicaban como funcionaban los relojes vectoriales y su implementación, ya que tuve bastante confusión entre las instrucciones y los outputs producidos.

En primer lugar, basado en la issue <https://github.com/IIC2523/2020-2-Tarea2/issues/18>, actualizo el reloj en solo 2 momentos: Cuando se va a enviar un mensaje, y se actualiza el elemento del vector correspondiente al proceso que lo envía, y cuando se procesa un mensaje recibido válido, y se cambia en un proceso sólo el valor del reloj correspondiente al sender.

También, por diseño, se ignoran las instrucciones de 'A' de auto-aumento del reloj tal y como indicaron en la issue, pues es probable que el reloj vectorial no funcione si se implementa.

Para implementar el algoritmo, hice una go function que se encargara de leer constantemente la queue, llamada `queue_manager()`, y en la queue iría almacenando los mensajes que llegaran junto con el valor de sus vectores. Así, para cada elemento de la queue, revisaría 1 por 1 los elementos del vector recibido, comparandolos con los elementos correspondientes del clock del proceso. Esto lo almacenaría en una variable llamada `difference_counter`, la cual incrementaría cada vez que un elemento del vector recibido fuera mayor que el mismo elemento del vector del reloj. Y si de los elementos recibidos hubiera exactamente 1 elemento mayor que los del clock del proceso, es decir, si `difference_counter` fuera igual a 1, entonces ese mensaje `RECVMSJ` sería procesado.



```
INCREMENTANDO RELOJ SEGUN MENSAJE ID 3
VIEJO RELOJ: [0 0 0 0 0]
NUEVO RELOJ: [0 0 0 1 0]
Popeando RECVMSJ 3 0 0 0 1 0
```

En caso de que hubiera 0, el mensaje se quedaría en la queue y nunca cumpliría con tener 1 elemento mayor que los del clock del proceso, por lo que nunca saldría de la queue, pues los cambios que especifica un mensaje así ya se encuentran procesados en el reloj vectorial, por medio, quizás, de la llegada de otros mensajes que también les fué multicasteado este mensaje.

En cambio, si tuviera más de 2 elementos mayores, significa que al proceso le faltaría que le llegara uno o más mensajes de otros procesos, dado que 1 o más procesos en algún momento enviaron un mensaje al sender, pero no a este proceso. De esa manera, si le llegara el mensaje

del proceso faltante, se actualizaría el reloj con este nuevo mensaje, y luego cuando se volviera a leer el mensaje que antes marcaba 2 elementos mayores que los del clock del proceso, ahora `difference_counter` marcaría 1, por lo que pasaría a ser ejecutado, y sería finalmente eliminado de la lista.

De esta manera, este contador de elementos mayores, implementado para cada proceso, y el cual se encuentra leyendo constantemente todos los elementos de la queue, me permite implementar el algoritmo, revisando así que se ejecuten los mensajes recibidos dentro de un proceso, siempre y cuando solo haya 1 dígito a actualizar en el contenido del vector recibido, comparado con el del proceso.

Cuando se procede a enviar un mensaje, se envía con el código "MSJ", con la ID del mensaje, y con el valor del reloj, al cual le fué incrementado el valor correspondiente a la ID del proceso antes de enviarlo.

```
Procediendo a mandar mensaje
Reloj vectorial viejo: [0 0 0 1 0]
Reloj vectorial nuevo: [0 0 0 1 1]
Mandando mensaje: MSJ 4 0 0 0 1 1 a proceso ID: 1
```

De esta manera, queda implementado el algoritmo Causally Ordered Multicast