Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2523 — Sistemas Distribuídos 2020-2

NOMBRE: Matías Duhalde
SECCIÓN: 1

Informe Tarea 2

1 Ejecución del programa

Como se pide en los requisitos de la tarea, esta consiste en dos archivos a ejecutar: el archivo del reloj lógico (`logical.go`), y el del reloj vectorial (`vectorial.go`).

El archivo principal del reloj vectorial está en el directorio `logical`. Para ejecutar el programa, se puede ejecutar el código base con:

```
go run ./logical.go <config_file> <id> <instructions_file>
```

El archivo principal del reloj vectorial está en el directorio `vectorial`. Para ejecutar el programa, se puede ejecutar el código base de manera similar con:

```
go run ./vectorial.go <config_file> <id> <instructions_file>
```

Ejemplo de ejecución para el programa de reloj lógico:

```
go run ./logical.go ../Ejemplo_Logico/config.txt 2 ../Ejemplo_Logico/instructions.txt
```

O se puede correr el ejecutable compilado (tal como pide el enunciado) usando:

```
./logical.go <config_file> <id> <instructions_file>
./vectorial.go <config_file> <id> <instructions_file>
```

Si es necesario volver a compilar el ejecutable, esto se puede hacer con:

```
go build -o logical
go build -o vectorial
```

2 Detalles generales del código

El código de esta tarea fue dividido en 4 partes principales. El paquete `ins` contiene las secciones relativas a las instrucciones (su lectura, parsing, y representación en structs). El paquete `comm` contiene código "común" para ambas implementaciones, incluyendo: la lectura y manejo del archivo de configuración, la representación de cada proceso, y operaciones para encontrarlos dentro de una lista. Los directorios `logical` y `vectorial` contienen las implementaciones de los relojes lógicos y vectoriales, respectivamente.

Estos últimos dos archivos contienen las operaciones de comunicación, paso de mensajes, coordinación, manejo de relojes, entre otras. La comunicación fue implementada con el protocolo TCP para el traspaso de mensajes, por lo que cada proceso se comportaría como servidor y cliente al mismo tiempo.

De estos archivos, las partes del código que recomiendo prestar más atención para la corrección serían las funciones:

- `executeInstructions`, la cual se encarga de tomar leer cada instrucción (las cuales se encuentran en forma de struct, definida en el paquete `ins`), y ejecutar la acción correspondiente.
- `handleMessage`, la cual está conectada a `listenClient` (que recibe mensajes), y se encarga de ejecutar los comportamientos correspondientes al mensaje recibido.
- `main`, que describe el flujo principal del programa.

3 Aspectos comunes de ambas implementaciones

En esta sección se explicarán los aspectos que comparten en común los dos programas: el reloj lógico y el vectorial.

3.1 Parte 1: Lectura de archivo de configuración

En primer lugar, al iniciarse un proceso, se leen los datos del archivo de configuración, el cual posee N procesos/líneas, y se revisa que este sea correcto. Una vez se obtiene la información de todos los otros procesos, se inician las conexiones.

3.2 Parte 2: Aceptar conexiones y conectarse a otros procesos

Un hilo se encarga de recibir/aceptar conexiones desde todos los otros procesos (`acceptConnections`), mientras que paralelamente se intenta conectar al resto de los procesos (`connectToHosts`). Dado que estos procesos pueden o no estar activos en este momento, el proceso realiza un número determinado de reintentos (por defecto, 10 por cada otro proceso) para intentar conectar con los otros procesos. Cuando el hilo encargado de aceptar conexiones recibe un proceso, genera otro hilo encargado de escuchar sus mensajes provenientes. Al alcanzar un número $N-1$ de conexiones entrantes, se deja de aceptar conexiones.

Cuando el hilo encargado de conectarse al resto de los procesos logra alcanzarlos a todos $(N-1)$, se termina este hilo. Las conexiones/sockets se guardan dentro de las estructuras de datos `Host`, las cuales se almacenan dentro de la array `otherHosts`.

Los procesos se identifican mutuamente enviando un string que contiene su address local (ver comando `ADDRESS`).

3.3 Parte 3: Lectura de archivo de instrucciones

Paralelamente a las dos tareas anteriores, se lee el archivo de instrucciones, y se almacenan como estructuras de datos (definidas en el paquete `ins`, y almacenadas en `instructionArray`).

3.4 Parte 4: Coordinación de partida

Cuando un proceso está listo, es decir, ha completado todas las tareas anteriores, pasa a esta parte (controlado por un `WaitGroup`), la cual avisa al resto de los procesos que está listo para empezar a ejecutar las instrucciones, enviando un comando `START` a todos. Una vez se reciben $N-1$ mensajes `START`, se tiene certeza de que todos los procesos están listos, y se comienzan a ejecutar las instrucciones.

3.5 Parte 5: Ejecución de instrucciones

En esta parte se van ejecutando las instrucciones previamente leídas. El comportamiento depende bastante del tipo de reloj, por lo que se explicará más adelante en el informe.

3.6 Parte 6: Coordinación de término

Al haber terminado un proceso sus propias instrucciones, envía al resto de los procesos el comando `END`. Este funciona de manera muy similar al comando `START`, en relación a que un proceso debe esperar a recibir $N-1$ mensajes `END` antes de terminar la ejecución.

Al terminar la ejecución, se muestra el clock final, y se cierran las conexiones.

4 Reloj lógico

En este código, el reloj lógico corresponde a un `int`, almacenado en la variable global `logicClock`. Este comienza en 0, y aumenta su valor en ciertas ocasiones.

Las instrucciones de aumento (tipo A), tienen el comportamiento pedido en el enunciado, es decir, se aumenta el clock en la cantidad especificada.

Cada vez que se encuentra una instrucción de enviar un mensaje (ya sea simple o multicast), este se aumenta en 1 y se procede a enviar el mensaje con el nuevo clock.

Cuando llega un mensaje, se compara el clock local al timestamp que el mensaje trae. Si se cumple que `logicClock < msg.Clock`, entonces se actualiza el valor del clock local, tal que `logicClock = msg.Clock`. Posteriormente, se aumenta el reloj local en 1, independiente si se cumple la condición anterior o no (así entendí el mecanismo de una issue).

5 *Totally-ordered multicast*

Para implementar *Totally-ordered multicast*, cada proceso posee una cola (queue) almacenada en la variable `messageQueue`. Cada mensaje tipo M (en el código, comando `MSG`), al ser recibido, es agregado a esta cola. Al mismo tiempo, al mensaje se le agrega una ID única, y se reordena la cola según el timestamp del mensaje (ver `sort.Slice` y `msgLT`).

Luego de agregar este mensaje, se envían a todos los procesos mensajes de acknowledgement (comando `ACK`), para informar al resto de la recepción. Este mensaje incluye la ID única del mensaje asignada por el proceso receptor, para poder identificar cual fue el mensaje reconocido. Cuando un proceso recibe un mensaje de `ACK`, este reenvía de vuelta un mensaje tipo `ACK_CONF`, que funciona como una manera de confirmar los acknowledgements.

Cada vez que un proceso recibe `ACK_CONF`, revisa cuantos de estos ha recibido para el mensaje de ID única k. Si ha recibido $N-1$ `ACK_CONFs` para este mensaje, luego revisa si el mensaje está al inicio de la cola. Si esto último se cumple, el mensaje se saca de esta, y se "ejecuta", que para efectos de esta simulación, esto solo significa hacer un print (ejemplo: `ID n - Executing message with ID: k`. Cuando se saca un mensaje de la cola, se revisa si el siguiente ya recibió todos los `ACK_CONFs`, para manejar el caso en que un mensaje que no estuviera al principio de la cola esté listo desde antes.

6 Reloj vectorial

En este código, el reloj vectorial corresponde a una array de `ints` de largo N, almacenada en la variable global `vectorialClock`. Todos sus elementos parten inicializados en 0, y su valor aumenta en ciertas ocasiones y dado ciertas condiciones.

Notar que según una issue, para que funcione bien el *Causally-ordered multicast*, se ignorarán las instrucciones de aumento del clock para este tipo de relojes (instrucción A). Si es necesario activar su funcionamiento, bastaría con des-comentar las líneas de código bajo `else if okInc` en la función `executeInstruction` de `vectorial.go`

Cada vez que un proceso se encuentra con una instrucción de enviar un mensaje (ya sea simple o multicast), aumenta su posición en su reloj en 1 (el proceso P_i aumenta $VC_i[i]$ en 1. Posteriormente, se envía el mensaje, con el clock ya aumentado.

Cada vez que un proceso "*entrega un mensaje a la aplicación*", es decir, recibe un mensaje y además este cumple las condiciones de causally ordered multicast, se actualiza el reloj de acuerdo al timestamp vectorial recibido, utilizando $max(\{ts(msg)[k], VC_j[k]\})$, siendo j el proceso que recibió el mensaje, para todo $k \neq i$, siendo i el proceso que envió el mensaje con el timestamp (ver siguiente sección para más información con respecto a este tema).

7 *Causally-ordered multicast*

Para implementar *Causally-ordered multicast*, nuevamente se hizo uso de una array. Causally-ordered multicast requiere que se cumplan 2 condiciones base para que se pueda ejecutar un mensaje recibido. Sea m el mensaje, P_i el proceso que envía el mensaje originalmente, y P_j el que lo recibe, entonces:

1. $ts(m)[i] = VC_j[i] + 1$

2. $ts(m)[k] \leq VC_j[k]$, para todo $k \neq i$

A nivel de código, estas condiciones se revisan en la función `verifyClock`. Ahora, al recibir un mensaje, se pueden producir 2 escenarios:

- que se cumplan las condiciones de *Causally-ordered multicast*, y
- que no se cumplan dichas condiciones

Para el primer caso, el flujo es bastante simple, el mensaje se pasa a la aplicación directamente (en esta simulación se representa mediante un *print* que dice `ID X - Executing message...`, y se actualizan las componentes del reloj vectorial según el timestamp recibido y la función max (explicado en la sección anterior).

Para el segundo caso los mensajes son añadidos a la array mencionada anteriormente (en la variable `messageArray`), junto con la ID del proceso que envió el mensaje, para poder ejecutarlo (pasarlo a la aplicación) posteriormente, una vez se cumplan las condiciones.

Para ejecutar los mensajes de la array (buffer), cada vez que se actualiza el reloj vectorial, se revisa si alguno de los mensajes de esta cumple la condición. Si es así, son ejecutados (representado por un *print* que dice `ID j - Executing message previously received from host i with message ID x...`), e inmediatamente sacados de la array. A nivel de código, la rutina `executeMessagesFromArray` se encarga de esto.

8 Consideraciones

- El programa se implementa usando TCP, por lo que no debería haber problema ejecutando el programa en múltiples máquinas concurrentemente.
- Para que hagan sentido las restricciones del enunciado, las IDs de los procesos, relojes, mensajes, entre otros, comienzan desde el 0.
- Dado que se utiliza TCP, los supuestos de *Totally-ordered multicast* se pueden garantizar (comunicación confiable y FIFO-ordered).
- No se implemento en ningún caso el "enviarse el mensaje a sí mismo" en multicast, no fue necesario en mi implementación.
- Según lo discutido en una de las issues, se desactivó la función de aumento de clock para el reloj vectorial, para evitar que se produzcan "deadlocks".