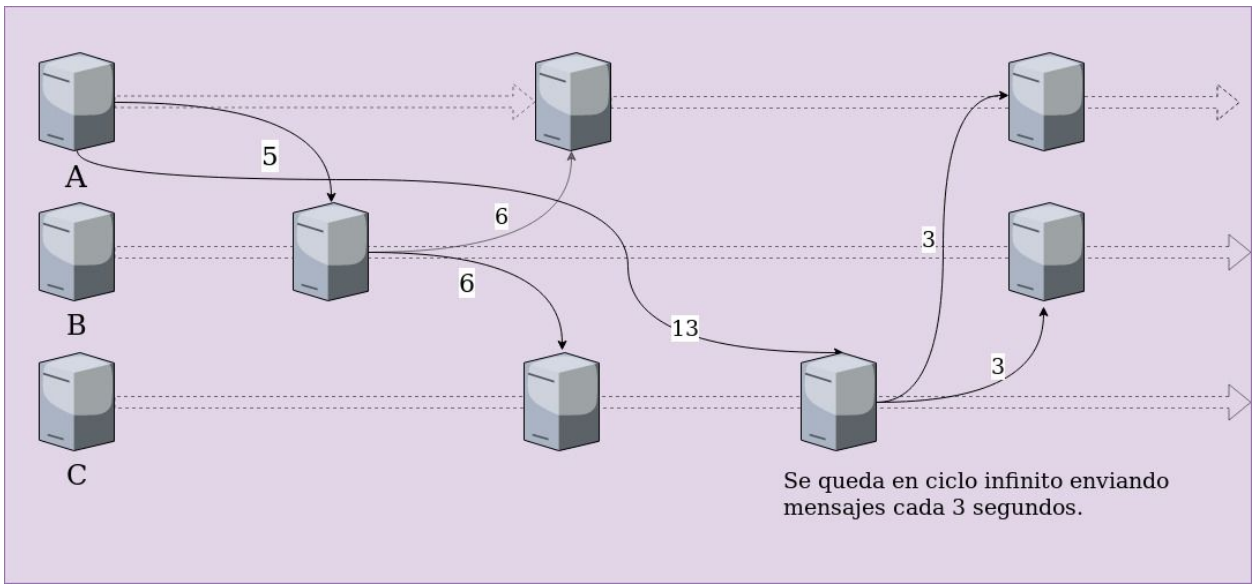
REDES Y SISTEMAS DISTRIBUIDOS
Práctica 1
Comunicación de grupo causal, relojes lógicos, estados globales
Profesor: Unai Arronategui

Introduccion

Esta memoria procederá a explicar el progreso alcanzado en el desarrollo de un sistema distribuido que basándose en una comunicación de grupo causal mediante la utilización de relojes vectoriales permita realizar el seguimiento de un modelo ejecutado de forma paralela a través de la red.

Solución e implementación

1. Comunicación de grupo causal



El diagrama presentado describe la interacción entre los 3 nodos de nuestro sistema distribuido a los cuales llamaremos como A, B, C al enviar mensajes para comunicarse donde cada mensaje posee una latencia asociada. Cada uno de los nodos posee un comportamiento específico para demostrar el

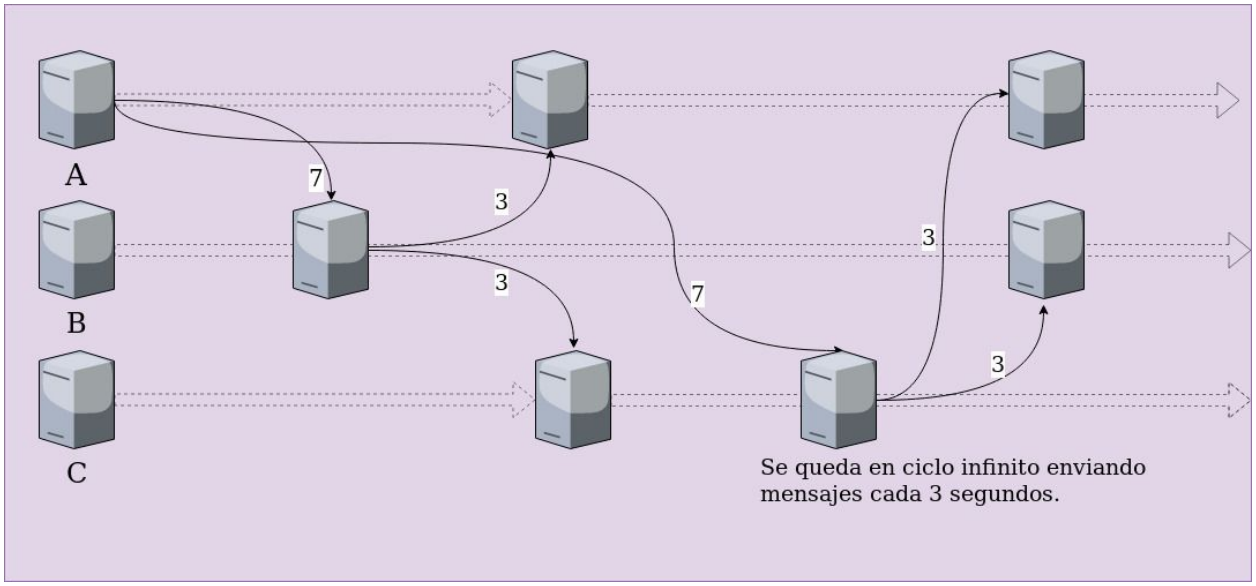
ordenamiento de los mensajes a través del uso de relojes vectoriales los cuales son representados con la estructura brindada junto al codigo de apoyo. En cuanto a la implementación, la firma de las principales funciones involucradas son:

```
func ReceiveGroup(localClock *v.VClock, addrs []string, port string,
messageList *[]MsgI, messageMap *map[string][]MsgI) {}

func SendGroup(msg MsgI, addrs []string, lats []string) {}

func SendToNode(msg MsgI, addr string, lat string) {}
```

Donde ReceiveGroup se encarga de actualizar el reloj al momento de recibir mensajes de tipo MsgI pertenecientes al grupo representado por addrs(sin contar su propia dirección) que han llegado por el puerto(port), además de esto se poseen messageList para la lista de mensajes recibidos que representan ejecución relevante del sistema distribuido y messageMap que es utilizada específicamente en el caso de la captura de estado del sistema, esta función es un ciclo infinito que siempre estará escuchando a la llegada de mensajes por TCP. SendGroup con SendToNode se encargan de enviar los mensajes a través de conexión directa TCP tomando en cuenta una latencia antes de enviar el mensaje el cual transmite el reloj vectorial local del nodo, la función de enviar a un nodo específico intentara hasta tres veces para enviar el mensaje en caso de que el nodo destino no esté escuchando o haya tenido algún fallo al momento de crear el conector, SengGroup es una creación de una goroutine por cada nodo al que se le debe enviar.

Para correr este ejemplo ejecutar go test -run TestSingle

2. Comunicación de grupo causal fiable con IP multicast



El diagrama presentado representa la interacción entre los distintos nodos del sistema distribuido desarrollado bajo el contexto de mensajería multicast y UDP. El diagrama sigue las mismas reglas de representación donde cada nodo al momento de enviar utiliza una latencia específica. Cada uno de los nodos posee un comportamiento específico e utiliza la siguiente firma de funciones:

```
func SendGroupM(msg MsgI, defaultMulticastAddress string, addrs
[]string, resendLat int, port string) {}

func receiveACKs(msg MsgI, addrs []string, port string, addrChecked
*map[string]bool) bool {}

func ReceiveGroupM(localClock *v.VClock, defaultMulticastAddress
string, port string, messageList *[]MsgI) {}

func sendACK(dest string, ori string) {}
```

Las funciones descritas tienen un comportamiento análogo a sus contrapartes con comunicación TCP pero al ser UDP un protocolo que no garantiza entrega de

mensajes se tuvo que implementar las funciones para enviar y recibir mensajes de tipo Ack que son representados con una estructura pertinente. SendGroupM realizará un envío de mensajes a través de la dirección de multicast y escuchara inmediatamente por la llegada de mensajes Ack, si estos no llegan antes de pasado el tiempo especificado por resendLat se volverá a enviar el mensaje a través de multicast, esta función es un ciclo infinito que siempre estará intentando enviar hasta recibir todos los Acks del mensaje enviado. La función ReceiveGroupM trabaja de la exacta misma manera que su contraparte de TCP con la diferencia de que al llegar un mensaje de operación es necesario enviar mensajes de ACK utilizando la función sendACK la cual crea una conexión directa a través de UDP hacia el nodo que originó el mensaje. Cada ACK es enviado una sola vez.

Para correr este ejemplo ejecutar go test -run TestMulti

3. Algoritmo Chandy-Lamport para determinar estados globales en sistemas distribuidos

Para la implementación de Chandy Lamport se utilizó el mismo ejemplo y conjunto de herramientas para el caso de TCP, las diferencias vienen en el mensaje enviado el cual ahora es representado con una estructura de nombre Marker además de aumentar la flexibilidad del ReceiveGroup para que manejara el caso de mensajes marcadores. Específicamente cosas que tiene que manejar el algoritmo es que al recibir un mensaje marcador este nodo debe encargarse de enviar nuevos mensajes marcadores al resto de los nodos del grupo al momento de recibir el primer mensaje marcador, los posteriores que lleguen serán anotados dentro de un mapa para poder identificar el momento en que el último mensaje marcador ha llegado y se debe imprimir el estado guardado. Además de esto es necesario tener un mapa de listas de mensajes recibidos por cada nodo donde se guardarán los mensajes de operación recibidos por cada canal de comunicación ya que estos representan los mensajes que se encontraban en vuelo después de recibir el primer mensaje marcador.

Para correr este ejemplo ejecutar go test -run TestSnap

4. Extras

El paquete de código incluye un archivo llamado Utilities.go el cual incluye todas las estructuras utilizadas además de las funciones de carpintería que se necesitaron, en particular las de ssh que se utilizan en la implementación de la ejecución a través de la red, los detalles específicos de direcciones, puertos, y rutas de ejecución deben ser alterados en el archivo para el correcto funcionamiento.