Universidad Zaragoza

Algoritmo Chandy-Lamport de estados globales

Redes y Sistemad Distribuidos

Práctica 1

Autor

Marius Sorin Crisan - 721609

ESCUELA DE INGENIERÍA Y ARQUITECTURA 2020-2021

Índice

1. Introducción

En este trabajo se lleva a cabo el diseño, implementación y validación de una librería para obtener estados globales de un sistema distribuido. La solución aportada se basa en el algoritmo de Chandy-Lamport [1]. Para la validación de la librería se utilizará *ShiHiz* [2], una herramienta de depuración distribuida, basada en relojes vectoriales[3]. La prueba definitiva se realizará en distribuido en tres máquinas del laboratorio 1.02 de la universidad.

2. Desarrollo

En esta sección se va a describir la arquitectura del sistema una vez desplegado en distribuido así como los tipos de mensajes que se envían entre los diferentes componentes y nodos del sistema. También se describe el comportamiento del sistema al enviar un mensaje a otro nodo y al obtener el estado global.

2.1. Arquitectura

Todos los nodos del sistema están conectados entre sí mediante una conexión punto a punto *tcp*. Por medio de esta conexión, se envían tanto los mensajes de la aplicación como la información relativa a la libraría desarrollada en esta práctica. La configuración/estructura del sistema se especifica en un fichero *json*. Este fichero contiene el nombre,*IP*, puerto, etc. de cada uno de los nodos que forman el sistema distribuido.

En la fase de diseño se tomó la decisión de desarrollar una librería que fuera lo más desacoplada posible de la aplicación principal. Como consecuencia, el software de cada nodo está compuesto por tres componentes bien diferenciados: aplicación, nodo o servicio de mensajería y el servicio *SnapNode* para la obtención de estados globales (Fig. 1).

La aplicación es el componente más simple de los tres ya que se trata de un servidor RPC que expone dos métodos: uno para enviar un mensaje grupal o individual (*SendGroup*) y el otro para iniciar el proceso de obtención del estado global (*MakeSnapshot*). Al finalizar este último, se escribe el estado global en el fichero de log correspondiente. Además, este componente cuenta con una *gorutina* que recibe los mensajes enviados por otros nodos hacia el nodo en cuestión y los registra en el fichero de log anteriormente mencionado para que el usuario tenga constancia de ello.

El componente *Node* funciona como un broker o servicio de mensajería.
Por una parte, escucha en un puerto tcp especificado por el usuario para recibir mensajes de otros nodos. La información recibida puede ser una marca, un mensaje de aplicación o el estado local de un nodo. En los tres casos envía esta información al componente *SnapNode* (canales *chRecvMsg_Mark y chRecvAllState*) ya que será utilizada por el algoritmo de obtención de estados globales [4]. Además, si se trata de un mensaje de aplicación este es enviado al componente *App* ya que será utilizado por

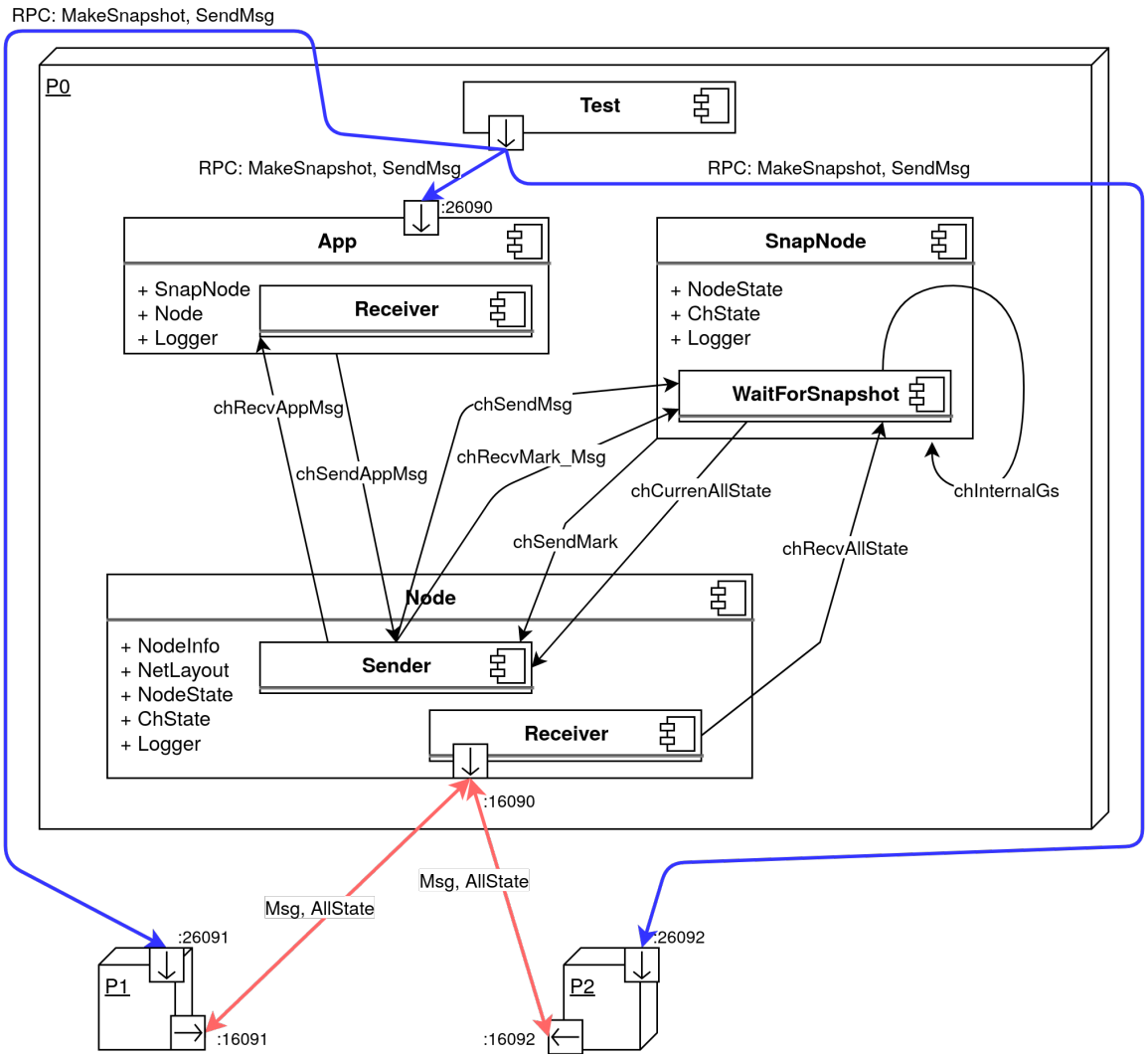

Figura 1: Diagrama de despliegue del sistema en tres nodos.

este (p. ej. para mostrarlo al usuario).

Por otra parte, el servicio *sender* es otra gorutina que sirve peticiones de la aplicación y del componente *SnapNode*. La única petición que se puede realizar desde la aplicación es enviar un mensaje a otro nodo (*chSendAppMsg*). Al recibir esta petición, se envía el mensaje al nodo correspondiente mediante la conexión *tcp* y también al *SnapNode*. Las posible peticiones generadas por *SnapNode* son enviar al resto de nodos una marca (*chSendMark*) o el estado local (*chCurrentState*).

Por último, el componente *SnapNode* es el encargado de llevar a cado el proceso de obtención del estado global. Recibe los mensajes enviados/recibidos, marcas y estados locales de otros procesos por medio de los canales *chSendMsg, chRecvMark_Msg y chRecvAllState*, respectivamente. Para notificar el envio del estado local o de una marca utiliza los canales *chSendMark y chCurrentAllState*.

Respecto a la depuración del sistema, se ha configurado un sistema de logging para todos los componentes de un mismo nodo. De esta forma, se puede registrar información sobre la ejecución del sistema con cuatro niveles de logging diferentes (*trace, info, warnning y error*). Al finalizar la batería

de test diseñada (Ver sec. 3), un script bash combina los ficheros generados en cada nodo en un solo fichero y ordena lo eventos en base a la hora local del nodo en el que se ha registrado dicho evento.

Respecto al tipo de datos enviados por la red, pueden ser un mensaje, una marca o el estado local de un proceso (Fig. 2). Toda esta información se envía por el mismo canal *tcp* utilizado por la aplicación. La librería *GoVector* transforma esta información al enviarla para añadirle el reloj vectorial utilizado para la depuración.

El mensaje está compuesto por el nombre del nodo remitente y el cuerpo del mensaje. La única diferencia entre un mensaje de aplicación y una marca es el cuerpo del mismo ya que el cuerpo de una marca contiene solamente la cadena "MARK".

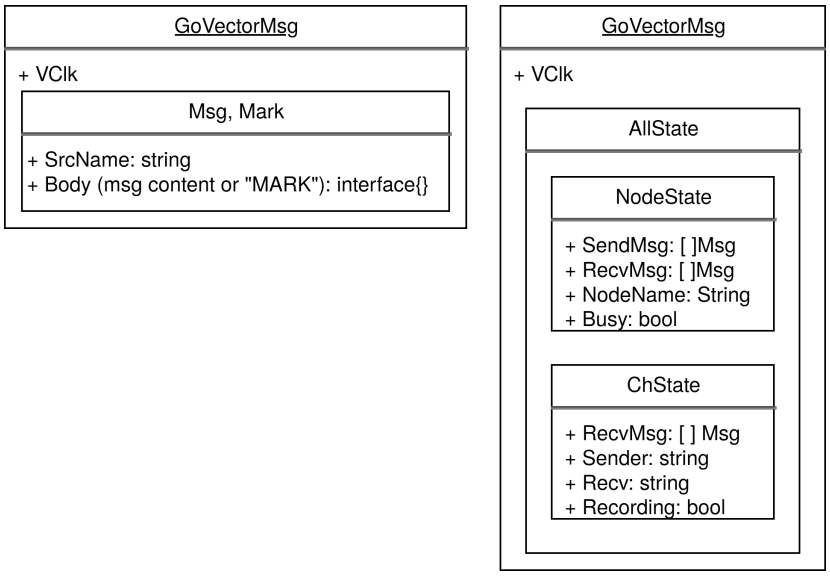

Figura 2: Tipos de mensajes enviados entre nodos.

Respecto al estado local, está compuesto por el estado del nodo y de los canales entrantes. Cada uno de los canales entrantes almacenan los mensajes recibidos desde que comenzó el proceso de obtención del estado global (*postrecording*). En cambio, en el nodo se almacenan los mensajes enviados y recibidos desde el último estado global y hasta el comienzo del proceso (*prerecording*).
Todos los mensajes registrados están almacenados de tal forma que pueda obtenerse el emisor y receptor de cualquier de ellos.

2.2. Comportamiento

En la figura 3 se muestra el diagrama de estados de cada uno de los componentes que hay en cada nodo. Respecto a la aplicación, el receiver (parte inferior) realiza una acción repetidamente: recibe mensajes y los guarda en el fichero de log. El hilo principal de este componente recibe peticiones RPC. Estas pueden ser enviar un mensaje u obtener el estado global. Al recibir esta última, se invoca al método *MakeSnapshot* del componente *SnapNode* para iniciar el algoritmo.

En el *SnapNode* al recibir la petición *MakeSnapshot* se guarda el estado actual del nodo, es decir, los mensajes enviados y recibidos desde el último *snapshot* y se empieza a grabar los mensajes recibidos por el resto de canales. A continuación, notifica al componente *Node* de que se debe enviar una marca (la primera) y se queda bloqueado hasta que recibe, mediante el canal *chInternalGs*, el estado global resultante.

La gorutina *WaitForSnapshot* espera mensajes del resto de nodos. Si es la primera marca que recibe guarda el estado del proceso local. Una vez recibidas las marcas de todos los nodos del sistema, envía su estado local al resto mediante *Node* y se queda esperando hasta recibir el estado local de todos los procesos. Finalmente, si se trata del proceso que inicio el algoritmo, envía el estado global al hilo principal.

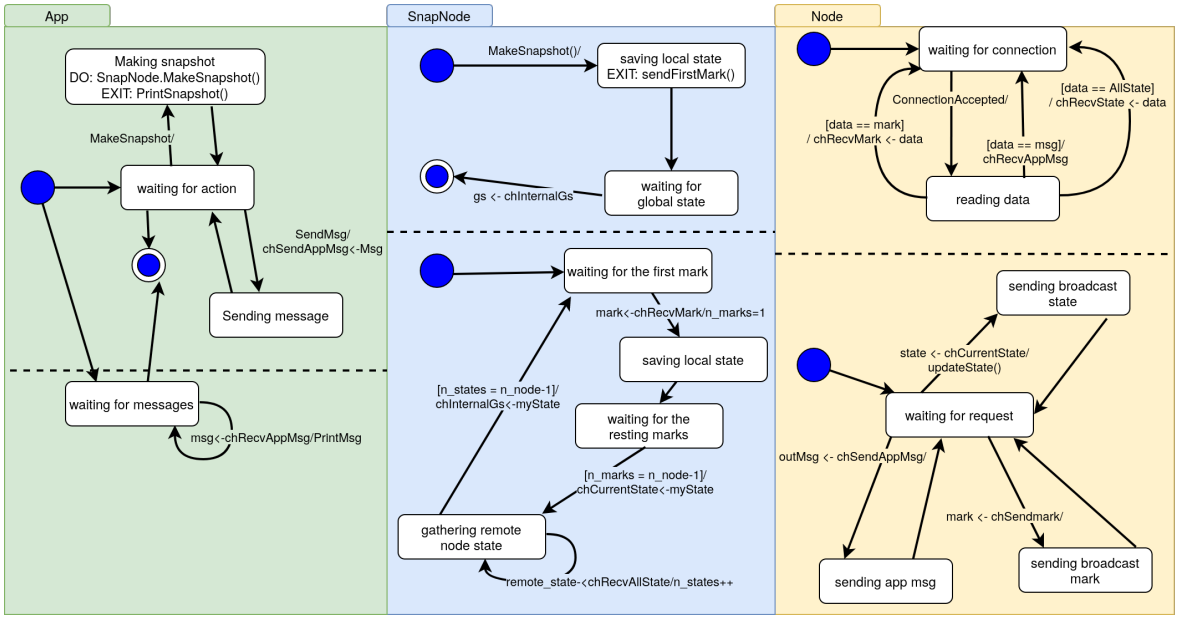

Figura 3: Diagrama de estados de un nodo del sistema. En la izquierda está el componente aplicación, en el centro el *Snapnode* y en la derecha el *Node*.

En cuanto al componente *Node*, la gorutina *Receiver* recibe mensajes de otros nodos y los procesa. Todos los mensajes recibidos (marcas, mensajes de aplicación y estados locales) son enviados al *SnapNode*. Además, si se trata de un mensaje de aplicación, se le envía a esta también.

La gorutina *Sender*, recibe peticiones tanto de la aplicación (enviar mensajes) como del *SnapNode* (marcas y estado local). Los mensajes son enviados solo a los nodos especificados en el mismo, mientras que las marcas y los estados deben ser enviados a todos los nodos del sistema.

3. Pruebas

Para ejecutar las pruebas desde una sola máquina y de una forma simple se hace uso de la librería RPC de go. Para ello, se ejecuta un binario de *testing* de go en una sola máquina. Este programa lee de un fichero *json* la información relativa a la estructura del sistema distribuido. A continuación, establece una conexión ssh con cada uno de los nodos y ejecuta la aplicación (componente *App*).

Pasado un tiempo de inicialización, la aplicación registrará un servidor RPC. Después, el programa de testing abre una conexión RPC con cada uno de los nodos. Estas conexiones se utilizan para enviar mensajes entre los nodos o iniciar el proceso de obtención de estados globales del sistema (Fig. 1). En todas las pruebas realizadas se utilizan tres nodos y se ha verificado su funcionamiento tanto en local como en 3 máquinas distintas del laboratorio 1.02.

En la primera prueba se obtiene el estado global sin ningún mensaje de aplicación enviado. En este caso se obtiene un estado global sin mensaje almacenados.

En la segunda prueba el nodo P0 envía un mensaje a P2 y después obtiene el estado global. Al finalizar esta prueba, el estado de P0 debe contener el mensaje como enviado y el estado de P2 como recibido. Con esta prueba se verifica que se almacena correctamente el estado de los nodos.

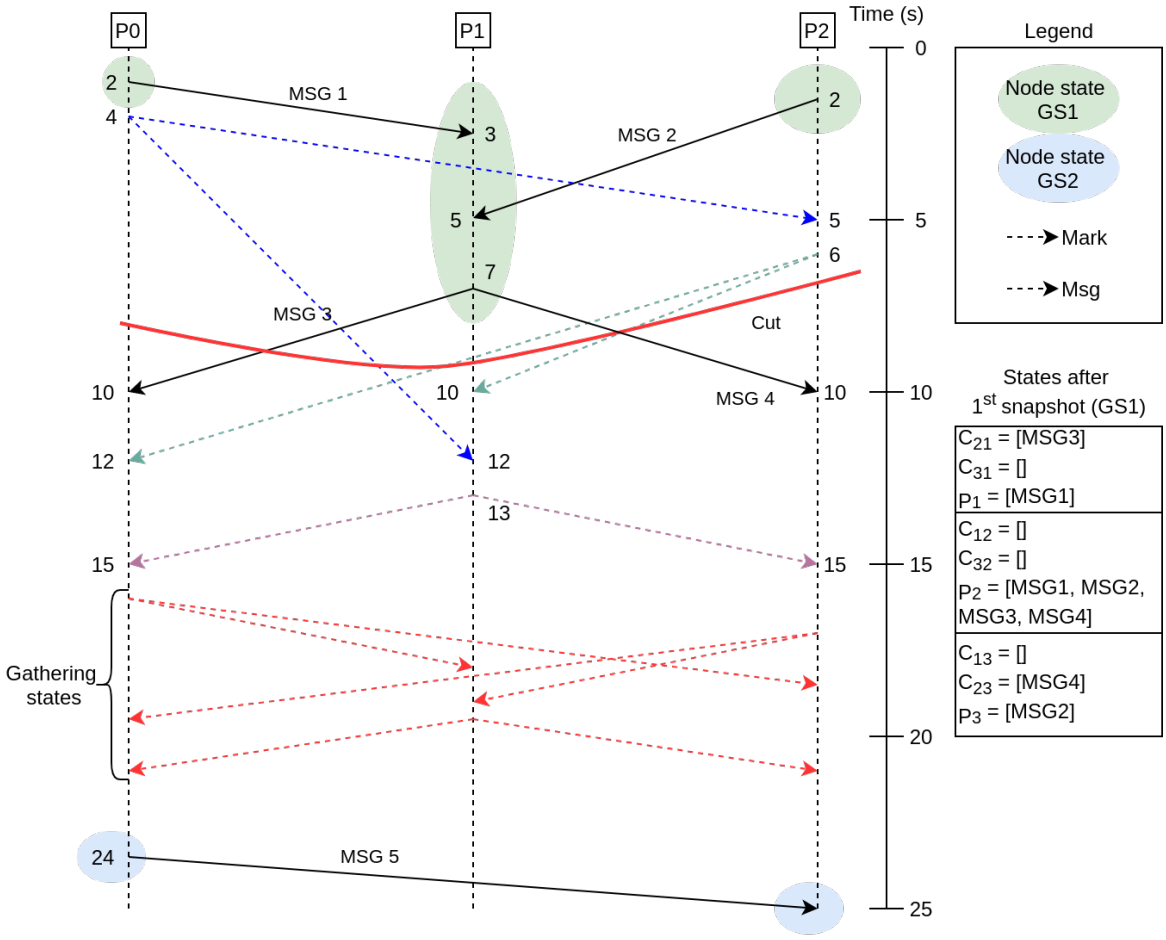

Figura 4: Caso de prueba con mensajes en tránsito durante la obtención del estado global.

La última prueba pretende verificar que los mensajes en tránsito durante la ejecución del algoritmo Chandy-Lamport se almacenan correctamente. Para ello, se propone conseguir la traza de ejecución de la figura 4. Los números que aparecen junto a los mensajes corresponden al tiempo transcurrido

desde el inicio del test. Estas estampillas temporales marcan los momentos aproximados en los que se debe recibir o enviar un mensaje de red. Para conseguir este escenario se introducen retardos en los mensajes. Por tanto, hay que tener en cuenta que la traza de ejecución de esta prueba puede variar en función de la saturación de la red.

Mediante los mensajes "MSG 1ƶ "MSG 2"se verifica de nuevo que se almacena bien el estado del nodo. Con los mensajes "MSG 3ƶ "MSG 4"se comprueba que se registra correctamente los estados de los canales al haber mensajes en tránsito. El primer estado global se muestra en el recuadro de la parte derecha de la figura 4. Al enviar enviar el mensaje *"MSG 5"* ya se ha obtenido el estado global. Finalmente se vuelve a obtener el estado global (no aparece en la gráfica) para verificar que se solo se almacenan en el estado local del nodo los mensajes que han ocurrido después del último estado global.

Observando la figura 5 se puede comprobar que la traza de ejecución de la prueba 3 es muy parecida a la pretendida (Fig. 4).

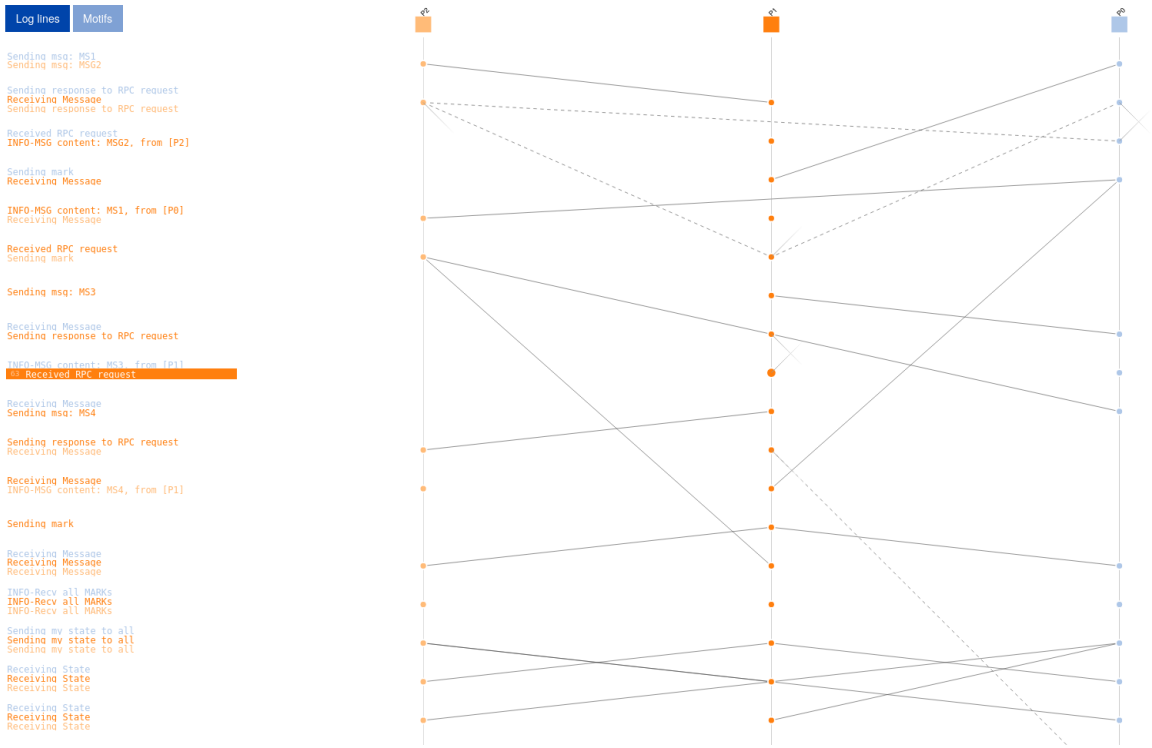

Figura 5: Traza ShiViz de la ejecución de la prueba 3.

4. Dificultades encontradas

La principal dificultad encontrada fue al compilar el programa ya que había un problema de incompatibilidad entre la versión de *GoVector* y la librería *msgpack* utilizada por este. Para solucionar se tuvo que cambiar la versión de la librería *msgpack* a una de diciembre 2018 y modificar en el código fuente dos funciones.

En cuanto al desarrollo de la solución, la mayor dificultad ha sido la sincronización de los mensajes enviados entre nodos para la tercera prueba realizada.

5. Conclusiones

En esta práctica se ha conseguido desarrollar una librería para obtener el estado global de un sistema distribuido. Esto ha ayudado a afianzar los conocimiento impartidos en la asignatura sobre relojes vectoriales, estados globales y depuración distribuida.
Durante el desarrollo de la misma se ha comprobado la utilidad que tiene una herramienta gráfica como *ShiViz* que te permita observar el tránsito de mensaje entre nodos ya que es mucho más fácil detectar un error que, por ejemplo, mirando los ficheros de log de cada nodo.

6. Referencias

[1] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[2] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging distributed systems: Challenges and options for validation and debugging. *Communications of the ACM*, 59(8):32–37, August 2016.

[3] Ivan Beschastnikh. Github.com/govector, October 2018.

[4] Lindsey Kuper. An example run of the chandy-lamport snapshot algorithm, April 2019.