Universidad Central de Venezuela
Facultad de Ciencias
Postgrado en Computación
Desarrollo de Aplicaciones Distribuidas

Francisco Sans

Robot Web Distribuido: Reporte Proyecto 1

1 Diseño de la Solución

La solución propuesta constará de un conjunto de componentes distribuidos que se encargarán de realizar el acceso a los enlaces suministrados. El código fuente del proyecto puede encontrarse ubicado en GitHub [1]. Para ello, se utilizarán containers de Docker para separar en un mismo entorno físico cada uno de los componentes como es mostrado en la Figura 1). La conexión de todas los componentes se apoya en la opción - *-net <nombre_red>* al momento de iniciar los containers, de tal manera que se crea una red con la cual se puede acceder a todos los componentes utilizando DNS con los nombres específicos de cada container.

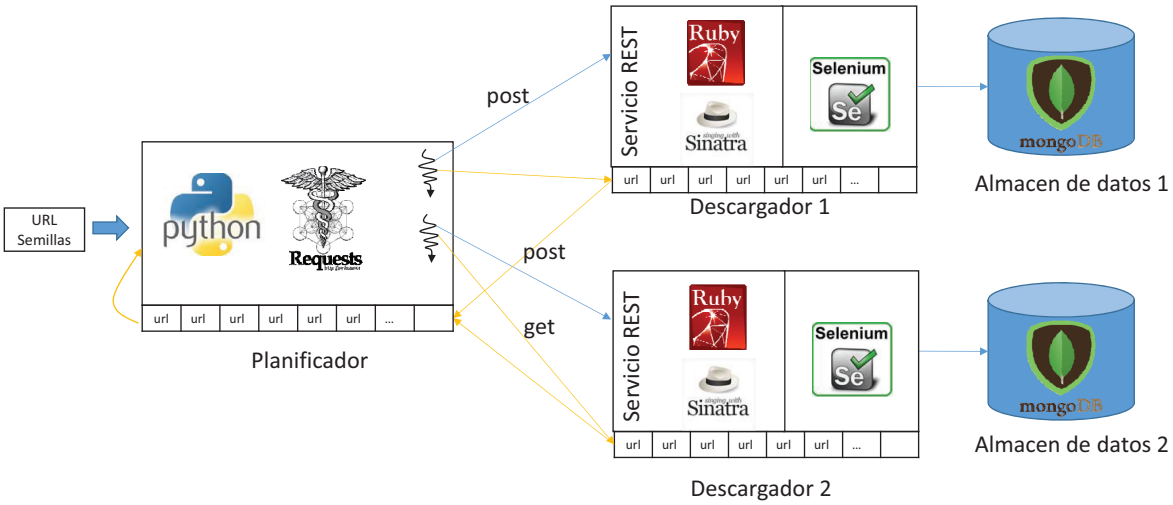

Figure 1: Esquema básico de la aplicación.

Para la comunicación entre el planificador y los descargadores se utilizó un servidor REST del lado de los descargadores, servicio que el planificador consumirá como cliente. En la programación el planificador se utilizó el lenguaje de programación Python junto con la librería Requests[2] para el consumo de los servicios. Se recibirá como entrada un archivo con los

[1] https://github.com/franjaviersans/DistributedSystemProject
[2] http://docs.python-requests.org/en/master/

enlaces semilla para empezar a inspeccionar los URL, además de un URL de domino base para filtrar la inspección de URLs, un entero máximo de URLs a visitar, y una lista de descargadores a los cuales conectarse. El cliente maneja un hilo por cada descargador disponible, donde cada hilo se encargará de consumir URLs de la cola hasta que no haya más cosas a procesar, o la cantidad máxima de URLs ya haya sido procesado. A través de un mensaje POST un hilo envía a su correspondiente descargador la dirección URL a procesar, y posteriormente obtiene en formato JSON los nuevos URLs encontrados a través de un mensaje GET al descargador. Se utilizan dos mensaje para evitar que haya un timeout por la espera de resultados. El mensaje GET podría retornar que el resultado todavía no está listo en el momento indicado, debido a que el descargador todavía está procesando la solicitud, por lo que se realiza un proceso de espera activa, mandando varios mensajes GET hasta obtener los resultados. Una vez obtenido los URLs de resultado, se procede a añadir en la cola aquellos elementos que no fueron añadidos anteriomente y que están dentro del dominio de filtrado. Posteriormente, el hilo vuelve a obtener un URL de la cola y repite el procedimiento. Los hilos terminarán su ejecución si ya procesaron la cantidad de URL máxima, o si ningún hilo tiene más URLs a procesar. Finalmente, los URL procesados con el estatus regresado por el servidor son almanceados en un archivo *log.txt* que puede ser accedido por el usuario al finalizar la ejecución del cliente.

Para los descargadores se utilizó el lenguaje de programación Ruby, que junto a la librería Sinatra[3], sirve para montar el servidor REST. Además, el descargador tendrá un servidor Selenium[4] Standalone para Firefox, el cual se encargará de hacer el renderizado de la página correspondiente al URL a procesar, para así poder obtener una imagen fidedigna de la misma. El servidor posee estados asumiendo que habla con un solo cliente, donde con el mensaje POST se obtiene el URL a procesar, y se lanza un hilo que se encargará de hacer la comunicación y el acceso a Selenium en segundo plano, de manera que el proceso sea no bloqueante. A partir del resultado obtenido por Selenium, el descargador obtendrá todos los enlaces contenidos en las etiquetas *<a>* y los almacenará en un arreglo para poder regresarlos al cliente. Utilizando el mensaje GET el cliente podrá obtener la respuesta correspondiente dependiendo de los siguientes tres casos:

- Se retorna un mensaje de error con el estatus 550 si hubo algún problema tratando de acceder al URL especificado por parte de Selenium. El mensaje de error es retornado en una cadena de caracteres para que el cliente conozca el error.
- Si el servidor todavía no está listo, envía un mensaje con la cadena *Not ready*, y el estatus 200.
- Si el mensaje está listo, se envía un mensaje en JSON con todos los URLs extraídos del URL inicial y estatus 200.

Finalmente los almacenes de datos fueron implementados con MongoDB como una base de datos NoSQL, que se utiliza para almacenar el código fuente de la página tal y como es

[3]http://sinatrarb.com/

[4]http://www.seleniumhq.org/

obtenido por Selenium. Las bases de datos pueden ser accedidas desde algún cliente mongo como es el caso de Robo 3T[5].

2 Scripts de Ejecución

Para facilitar la instalación, ejecución y prueba de la solución, se presentan un conjunto de scripts que contienen un conjunto de comandos para realizar estas tareas de manera automática. Los *scripts* tienen el siguiente uso:

- install_containers.sh: se encarga de la ejecución de dos DOCKERFILEs, correspondientes a: (1) la imagen de Python instalando la librería Requests para realizar consultas REST; y (2) la imagen del Selenium Standalone con Firefox, instalando Ruby con las librerías *sinatra*, *selenium-webdriver*, *json* y *mongo*. Además, este script también se encarga de descargar la imagen de mongo sin ningún complemento adicional.

- start_containers.sh: en este script se inician las imágenes de los descargadores, de las bases de datos y de la red de comunicación. Para las bases de datos se crean directorios para almacenar la información de las páginas. Los descargadores correrán el archivo ruby con el nombre *serverRest.rb*, el cual tiene toda la lógica del servidor. El script recibe por parámetro un número entero que indica la cantidad de descargadores que quieren iniciarse, y el nombre de cada uno con sus respectivas bases de datos son almacenados en archivos, de tal manera que el cliente pueda accederlas fácilmente.

- start_client.sh: inicia el container del planificador, que se encargará de realizar la tarea del robot. El planificador correrá el archivo *client.py*, el cual contiene toda la lógica del cliente. El cliente leerá los descargadores disponibles del archivo generado por el script anterior.

- kill_containers.sh: es utilizado para remover de Docker los containers utilizados para el planificador, los descargadores y las bases de datos. Adicionalmente, también se elimina la red creada para la communicación y los directorios que contienen las bases de datos, de tal manera de limpiar la computadora de la ejecución del programa.

[5] https://robomongo.org/