Email Based Middleware

Alexander A. González Fertel, Sandor Martín Leyva
Universidad de la Habana
Facultad de Matemática y Computación
alexfertel97@gmail.com, s.martin@estudiantes.matcom.uh.cu

Resumen

Este artículo es parte del proyecto Middleware Basado en Emails de la asignatura Sistemas Distribuidos. En dicho documento explicamos la parte teórica del funcionamiento del proyecto.

1. Arquitectura

1.1. Cliente - Servidor

Como fue orientado en el proyecto, *EBM* consta de dos componentes, una biblioteca *ebmc* como cliente y una aplicación servidor.

El flujo es el siguiente: Una aplicación cliente, digamos *sender-app* (La aplicación que proponemos como prueba de concepto), importa *ebmc* y a través de dicha biblioteca, luego de proveer un cliente de correo válido y una dirección de correo de alguno de nuestros servidores, empieza a interactuar con nuestro sistema como si este fuera una entidad única.

Esto facilita la abstracción de la aplicación cliente de la comunicación con otro servicio que use *ebmc*, permitiendo al usuario a centrarse en la lógica de su programa.

1.2. Cliente

En el archivo *README.md* de nuestro proyecto se encuentran las instrucciones para instalar *ebmc*, el cual expone la clase *EBMC*.

Los métodos *register*, *login*, aseguran que un usuario del middleware pueda acceder a sus datos usando cualquier correo, siempre que provea de su identificador correctamente. El método *register* registra a un usuario que nunca se ha conectado, mientras que el método *login* autentica al usuario en el sistema, devolviendole un *token* para usar en cada interacción con el sistema, simulando la autenticación de las arquitecturas RESTful y por tanto, su seguridad. Vale notar que dicho *token* es la misma llave que mantiene la información del usuario en el sistema (Una llave de *chord*), lo que hace que el usuario tenga ubicuidad con respecto a su correo, si un usuario conoce su *token* puede utilizar indistintamente cualquier correo válido.

La clase *EBMC* tiene además los métodos *send* para comunicación entre pares, *publish*, *subscribe* y *unsubscribe* para soportar *PUB/SUB* y *create event* para crear .^{ev}entos.^{a} los cuales los usuarios se pueden suscribir, los cuales no son más que mapeos entre usuarios y *strings*.

1.3. Servidor

Nuestro servidor es una aplicación inicializada por un *script*, escrito en *Python*, que expone mediante *python-fire* la clase *EBMS*. Dicha clase está asociada a un servidor de correo mediante una respectiva cuenta de correo y además esta asociada a una dirección de la capa de transporte, mediante la cual se comunica con demás instancias de dicha clase. Toda comunicación cliente-servidor ocurre mediante correos electrónicos.

Para facilitar la puesta en producción de dicho servidor, hemos dockerizado completamente la aplicación y explicamos la manera de ejecutarla en el archivo *README.md*.

1.4. Chord

Para lograr los requerimientos de tolerancia a fallas, hemos decidido usar el sistema basado en tablas de hash distribuidas *chord*, asegurando además escalabilidad, eficiencia, balance de carga y disponibilidad en nuestro sistema.

Cada instancia de la clase *EBMS* es un nodo de *chord* y por tanto tiene un identificador, el cual es generado (junto con todo el espacio de llaves de nuestro sistema) usando la función de hash *SHA-1*, ya que por definición las direcciones de correo electrónico son únicas, dicho identificador también es único.

Dicha clase es también la encargada de replicar la información basándose en la propuesta de los autores de *chord*, manteniendo r sucesores con réplicas de los pares llave/valor.

2. Comunicación

Al utilizar como transporte para comunicar dos aplicaciones correos y querer garantizar orden y confianza en la entrega de los mensajes, lo más lógico sería empezar comparándolo con la capa de transporte actual de las redes. Dicha comparación no es con *UDP*, puesto que no cumple nuestros requisitos.

2.1. Transporte

Un *socket* define una abstracción a nivel de aplicación para recibir y enviar mensajes, a traves de un *socket* podemos enviar cualquier cantidad de información que mientras sea soportado se enviará y recibirá con seguridad, al usar el protocolo *TCP*, a pesar de estar sobre un medio de transporte no confiable, como lo son las capas bajas de las redes.

Nosotro debemos garantizar lo mismo, pero ahora nuestro medio de transporte no confiable son los correos electrónicos. Como tal, los correos definen una interfaz equivalente a un *socket UDP*, por tanto, nos centramos en tratar de convertir dicho *socket* en uno *TCP*.

2.2. Bloques y Mensajes

Nuestra aplicación convierte la información que se desea enviar en instancias de la clase *Message*. Es decir, a la hora de enviar un mensaje, cualquiera que este sea, lo convertimos en una instancia de *Message*, asignándole un identificador que refleja su orden (tiempo actual). Además, mantenemos una cola de mensajes en cada parte de la comunicación tal que antes de enviar y al recibir ordenamos los mensajes. Esto garantiza el correcto orden de los mensajes si asumimos que todos llegan al destinatario, lo cual sabemos que es improbable, sino imposible. El protocolo *SMTP* no garantiza confiabilidad a la hora de entregar los correos, lo cual quisimos arreglar justo como *TCP*, manteniendo una ventana (window) y respondiendo a los mensajes con *acknowledgements* o *ACKs*, pero no pudimos.

Nos enfrentamos a otro problema, los correos tienen un tamaño limitado, por lo cual, si nuestro mensaje tiene longitud mayor a la soportada por el servidor de correo, no podemos enviarlo. Como solución definimos una nueva clase *Block* de forma tal que todo *Message* esta formado por una lista de bloques, y en realidad son los bloques (al heredar de *email.message.EmailMessage*) los verdaderos correos electrónicos que envía nuestro middleware.

2.3. Broker

En la literatura (y en economía), un *broker* es una entidad que arregla transacciones entre 2 o más partes, nosotros definimos la clase *Broker* que se encarga del manejo de todo el trabajo con los mensajes, bloques y correos electrónicos y de la cual tanto *EBMC* como *EBMS* mantienen una instancia asociada.

Dicha clase mantiene un hilo leyendo del servidor de correo a través del protocolo *IMAP*, mantiene otro hilo procesando 2 colas, una de configuración y otra de datos y además procesa todos los bloques y los empareja con sus respectivas instancias de *Message* traduciendo de correo electrónico a *Block*.

3. Estructura

Hablemos un poco de la estructura de nuestros servidores y los protocolos que definimos.

3.1. Asunto

Si un usuario desea utilizar nuestro middleware, primero debe registrarse e iniciar sesión en nuestra aplicación, para luego realizar cualquier acción que oferta el middleware. Notemos que los mensajes de registro solo interactúan con el servidor asociado a dicho cliente y luego de ser procesados el servidor interactúa con el sistema, esta acción es diferente a, por ejemplo, hacer una publicación, que requiere resolver los ususarios suscritos al que publica y luego enrutar correos a cada uno de los suscriptores.

Por tanto se hace necesario definir protocolos para así multiplexar las acciones de nuestro servidor. Nuestra solución se basa en un formato bien conocido haciendo uso de los asuntos de los correos. Un *subject* o asunto, es un diccionario que usando la biblioteca *json* convertimos a texto y enviamos en cada asunto de cada bloque(correo), dicho asunto contiene "metadatos", sobre el bloque y resuelve el multiplexado de los mensajes.

Definimos dos tipos de mensajes y por tanto dos colas de mensajes, mensajes de configuración y mensajes de datos. Los mensajes de configuración cambian la información subyacente en *Chord*, mientras que los mensajes de datos, envían información de un cliente a otro.

3.2. Enrutado

Para enviar un mensaje, podemos pensar en cómo es mejor, desde el punto de vista de eficiencia, resolver el destinatario, que dado que un usuario no está identificado por una dirección de correo, es un problema, además, deseamos minimizar la cantidad de correos. Una solución sencilla e intuitiva a este problema, y es una de las razones por las que surge la necesidad de usar *Chord* (o algún protocolo de almacenamiento de datos, consistente), es guardar un mapeo entre identificadores y la dirección de correo por la que responde el usuario actualmente.

Esto garantiza que para enviar un mensaje solo se necesiten tres direcciones de correo, el usuario que desea enviar el mensaje, el servidor asociado al usuario y el destinatario, el cual es resuelto en $O(logN)$ gracias a *Chord*. Se envían además solo tres mensajes (asumiendo que no se ha implementado la funcionalidad de los ACK), un correo de configuración del *ebmc* al servidor de correo asociado a dicho usuario, un correo de vuelta al usuario con el correo del destinatario resuelto y finalmente, un mensaje al destinatario, donde por mensajes decimos todos los bloques que sean necesarios para hacer llegar el contenido del mensaje al destinatario.

4. Tecnologías

Para la implementación del proyecto se utilizó *Python 3.6*. La biblioteca *imbox* se utilizó para el manejo de las conexiónes a servidores *IMAP*, esta biblioteca es muy simple de usar y facilita mucho el parseo de los correos electrónicos. Para la implementación de *Chord*, utilizamos *rpyc Remote Python Call*, una biblioteca para el trabajo con objetos remotos (RPC). Usamos *docker* para facilitar el desarrollo y puesta en producción de la aplicación y *python-fire* para facilitar la inicialización del servidor desde una *CLI*. Trabajamos con *smtplib* para el envío de correos.