UNIVERSIDAD DE ANTIOQUIA

Spam Detection Bridge

Autor(es)

Daniel Uribe Giraldo

Universidad de Antioquia

Facultad de ingeniería, Departamento de sistemas

Medellín, Colombia

2020

Spam Detection Bridge

Daniel Uribe Giraldo

Tesis o trabajo de investigación presentada(o) como requisito parcial para optar al título de:

Ingeniero de sistemas

Asesores (a):

Gabriel Dario Uribe Guerra

Juan Pablo Vergara Villarraga

Universidad de Antioquia

Facultad de ingeniería, Departamento de sistemas.

Medellín, Colombia

2020.

RESUMEN

Para las empresas que prestan servicios de telecomunicaciones, en este caso específico Analítica de datos, siempre ha sido un reto detectar y actuar ante llamadas de spam y/o con fines maliciosos, por esta razón la compañía Marchex decide desarrollar un conjunto de servicios tecnológicos para resolver esta problemática. Estos servicios están conformados por un módulo encargado de recibir las llamadas; De ahora en adelante nos referiremos a este como Call-Stack. Unos modelos entrenados con técnicas de aprendizaje de máquinas para detectar posibles llamadas de SPAM y/o llamadas maliciosas, y un módulo encargado de conectar el Call-Stack y los modelos encargados; De ahora en adelante nos referiremos a este módulo como SPAM DETECTION BRIDGE.

En este documento nos centraremos en desglosar el módulo de SPAM DETECTION BRIDGE mostrando las tecnologías usadas para desarrollarlo, arquitectura, patrones, protocolos de transporte, etc.

INTRODUCCIÓN

Existen muchas compañías que ofrecen una amplitud de servicios relacionados con telecomunicaciones, entre estas empresas se encuentra Marchex, que entre la variedad de servicios que ofrece se encuentra el servicio de Call Analytics. Con el fin de ofrecer este servicio de gran calidad, Marchex se tiene que asegurar de afrontar y manejar todos los posibles desafíos tecnológicos que hay actualmente, entre estos retos se encuentra el correcto manejo de llamadas maliciosas y/o llamadas de SPAM. Con el fin de afrontar este reto tecnológico, Marchex ha decidido desarrollar un módulo conformado por un conjunto de servicios encargados de detectar oportunamente estas llamadas de SPAM y/o maliciosas y darles un correcto manejo a estas mismas.

Tomando como premisa usar soluciones vanguardistas, Marchex decide dividir este reto en tres principales frentes: Un Call-Stack (preexistente), Modelos estadísticos que fueron entrenados con información preexistente de la misma compañía con el fin de detectar llamadas de spam y/o maliciosas, y un servicio intermedio encargado de unir los dos proyectos anteriores(Spam Detection Bridge).

OBJETIVO GENERAL

Desarrollar un puente entre el Call-Stack y los modelos estadísticos de detección de spam y/o llamadas maliciosas.

OBJETIVOS ESPECÍFICOS

- Desarrollar un servidor SIP que permite recibir invitaciones(INVITE, OPTIONS, BYE)
- Permitir recibir paquetes UDP por el protocolo RTP haciendo uso del servidor SIP.
- Convertir los paquetes de audio a formato WAVE.
- Enviar los paquetes de audio en formato WAVE a un servicio externo a través de HTTP/2.0(Streaming)
- Permitir recibir a través de HTTP/1.1 la predicción proveniente de un servicio externo
- Enviar la predicción a una pila de llamadas a través de HTTP/1.1

DESARROLLO DE SPAM DETECTION BRIDGE

EQUIPO DE DESARROLLO

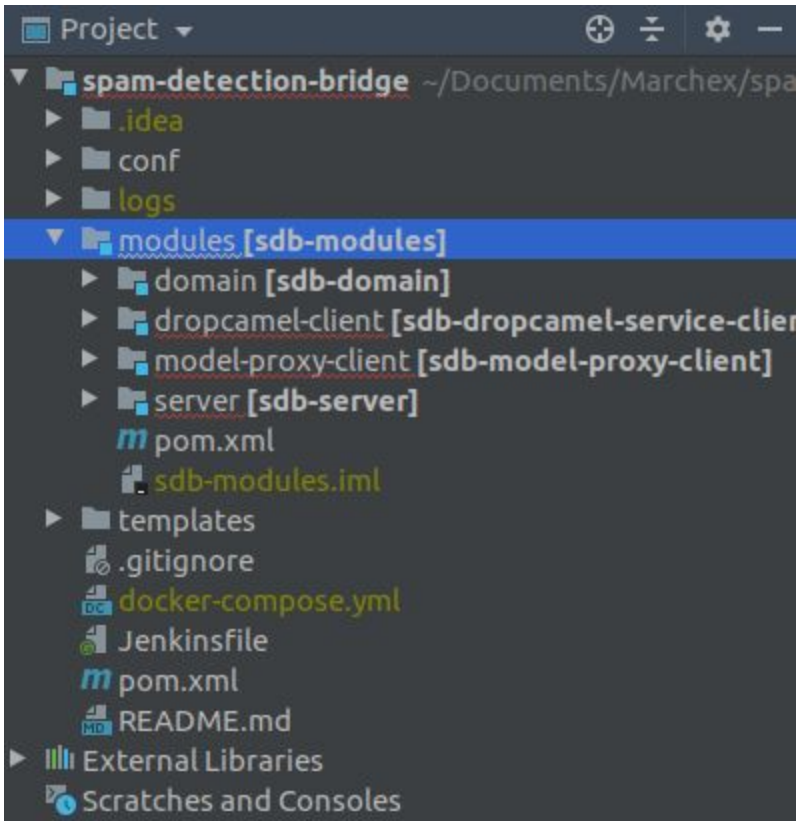
El equipo de desarrollo para este proyecto estaba conformado por:
Daniel Uribe Giraldo (mid-level developer)
Anand Raghavan (Senior developer)
Dado que el rol de Anand Raghavan para este proyecto estaba enfocado únicamente en la revisión de Pull Request, casi la totalidad del código fue escrito por Daniel Uribe Giraldo.

METODOLOGÍA ÁGIL

El proceso de desarrollo de spam detection bridge se ha llevado a cabo bajo el marco de trabajo SCRUM , el cual se enfoca en trabajos pequeños, entregas rápidas y constantes, y buena comunicación con el cliente.
Al inicio del proyecto se plantearon unos objetivos con el product owner (PO) y el equipo de desarrollo, también se realizó la definición y priorización de las historias de usuario, se acordaron sprints cada semana y la retroalimentación de los usuarios de este servicio durante el proceso.

VISIÓN GENERAL DEL PROYECTO

Debido a la infraestructura y la base tecnológica de la compañía de Marchex, se opta por desarrollar un microservicio en Java 11 con Spring Boot, usando como gestor de dependencias Maven.

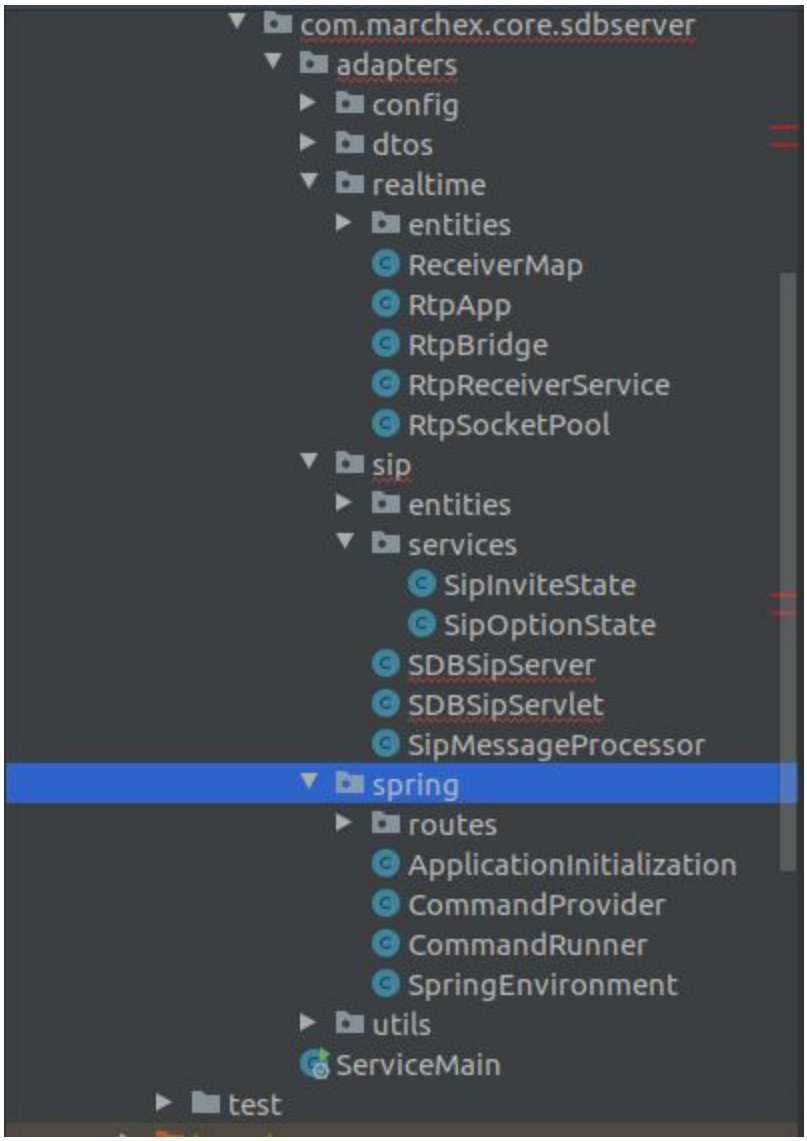


COMPONENTES DEL PROYECTO

- **SIP + RTP SERVER:**

Se implementó un servidor SIP debido a que el Call-Stack usa estas tecnologías para comunicarse con los demás servicios, adicionalmente añadimos el protocolo de transporte RTP. El cual es muy usado en conjunto con el protocolo SIP, para recibir los paquetes de audio provenientes del Call-Stack.

Este módulo consiste en un servidor SIP con la finalidad de tener comunicación directa con el Call-Stack, ya que el Call-Stack se comunica de esta manera. Cuando el servidor SIP recibe un INVITE (petición específica del protocolo SIP) este acepta la invitación y responde con dos puertos UDP(obtenidos mediante un pool de puertos desarrollado dentro del proyecto) por los cuales el Call-Stack iniciará el envío de paquetes RTP mediante los puertos expuestos. De manera resumida, con este módulo obtenemos un servidor SIP, que está habilitado para recibir peticiones y mediante el protocolo de transporte RTP obtener paquetes de audio por medio de puertos UDP.



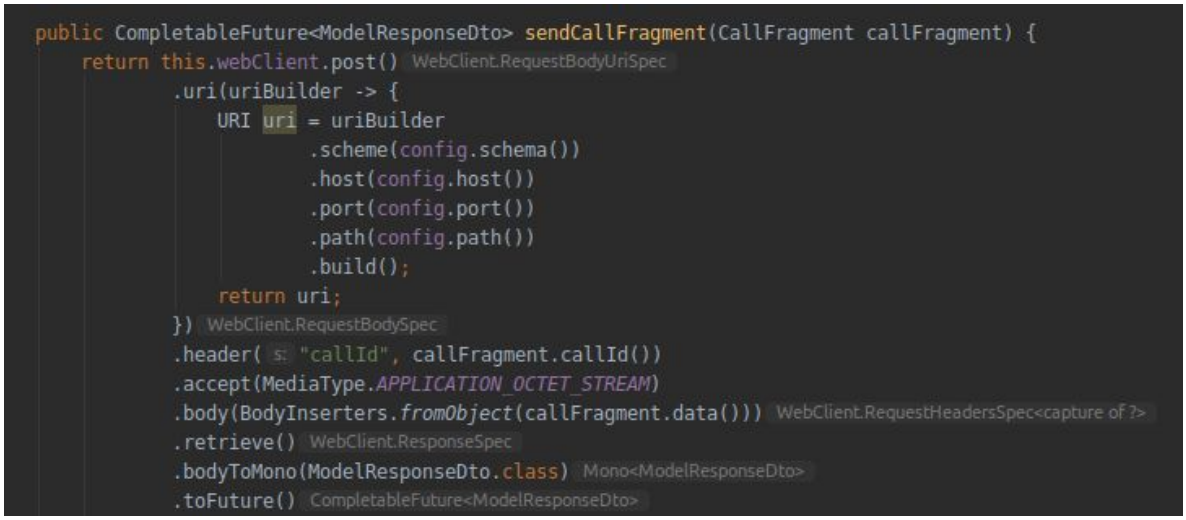

```
public void initialize() {
    this.udpConnector.setPort(sipPort);
    this.udpConnector.setHost(hostIP);
    this.cipangoSipServer.setConnectors(new SipConnector[]{udpConnector});
    this.sipServletHolder = sipAppContext.getServletHandler().addServlet(SDBSipServlet.getClass().getName());
    this.sipServletHolder.setServlet(SDBSipServlet);
    this.sipServletHolder.setInitParameter( param: "hostIP", hostIP);
    cipangoSipServer.setHandler(sipAppContext);
    sipAppContext.getSessionHandler().getSessionManager().setSessionTimeout(sessionTimeout);

    Try.run(() -> {
        cipangoSipServer.getThreadPool().join();
        cipangoSipServer.start();
    }).onFailure(error -> log.error("Error starting Cipango server: {}", error.getCause().getMessage()));
}
```

- **CLIENTE HTTP:**

Se implementa un cliente HTTP para enviar los paquetes de audio a los modelos estadísticos.
Librería usada: org.springframework.web.reactive.function.client.WebClient

Este módulo consiste en un cliente HTTP el cual será el encargado de enviar el audio en formato WAVE a los modelos estadísticos que posteriormente retornarán un JSON correspondiente a la respuesta de los modelos, los cual indicará si el audio enviado corresponde o no a un audio malicioso o audio de SPAM.



- **CLIENTE HTTP:**

Se implementa un cliente HTTP adicional para enviar la respuesta de los modelos al ente encargado de realizar alguna acción sobre la llamada.
Librería usada: org.springframework.web.reactive.function.client.WebClient

Este módulo consiste en un cliente HTTP, el cual está encargado de enviar un JSON con la respuesta de los modelos estadísticos al ente encargado de tomar decisiones sobre las llamadas en curso, estas decisiones pueden ser: colgar la llamada, reproducir un archivo IVR, o dejar que la llamada se dé con normalidad.

```java
public CompletableFuture<String> sendPrediction(String host, PredictionDto prediction) {
    ObjectMapper objectMapper = new ObjectMapper();

    return this.webClient.post() WebClient.RequestBodyUriSpec
            .uri(uriBuilder -> {
                URI uri = uriBuilder
                        .scheme(config.schema())
                        .host(host)
                        .port(config.port())
                        .path(config.path())
                        .build();
                log.info("Sending prediction to Dropcamel URL: {}", uri);
                return uri;
            }) WebClient.RequestBodySpec
            .accept(MediaType.APPLICATION_JSON)
            .body(prediction) WebClient.RequestHeadersSpec<capture of ?>
            .retrieve() WebClient.ResponseSpec
            .bodyToMono(String.class) Mono<String>
            .toFuture() CompletableFuture<String>
```

- **DOMINIO:**

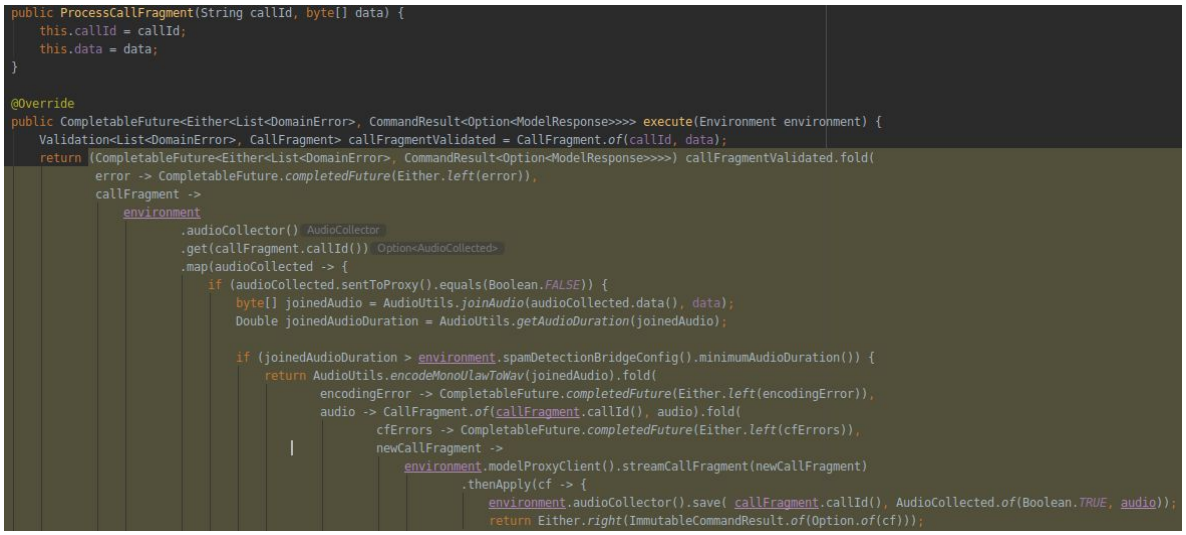
Se implementa un módulo de dominio encargado de realizar todas las validaciones necesarias sobre los paquetes de audio enviados desde el Call-Stack, cambiar el formato de audio de Mono Ulaw a WAVE, y por último este módulo también está encargado de ser el intermediario entre todos los otros módulos. esto debido a decisiones de arquitectura que se explicarán más adelante.

```java
public ForwardPrediction(String callId, Double score, String type) {
    this.callId = callId;
    this.score = score;
    this.type = type;
}

@Override
public CompletableFuture<Either<List<DomainError>, CommandResult<DropcamelResponse>>> execute(Environment environment) {
    Validation<List<DomainError>, CallInfo> callInfoValidation = CallInfo.of(callId);
    Validation<List<DomainError>, Prediction> predictionValidation = Prediction.of(score, type);
    Validation<List<DomainError>, CallInfo> callInfoByIdValidation = environment.callInfoProvider().getCallInfo(callId).toValidation();

    return Validation.combine(callInfoValidation, predictionValidation, callInfoByIdValidation)
            .ap(Tuple3::new)
            .fold(
                    errors -> CompletableFuture.completedFuture(Either.left(errors.toList().flatMap(Function.identity()))),
                    tuple -> {
                        Validation<List<DomainError>, CallInfo> callInfoValidated = CallInfo.of(tuple._1.callId(), tuple._3.hostName().getOrElse( other: ""));

                        return callInfoValidated.fold(
                                error -> CompletableFuture.completedFuture(Either.left(error)),
                                ci ->
                                        environment
                                                .dropcamelClient()
                                                .sendModelPrediction(ci, tuple._2)
                                                .thenApplyAsync( dropcamelResponse ->
                                                    Either.right(ImmutableCommandResult.of(dropcamelResponse))
                                                )
                        );
                    }
```



ARQUITECTURA

Se decide desarrollar un microservicio usando como enfoque de desarrollo DDD (Domain Driven Design), esto debido a la importancia de separar la lógica del dominio de los agentes externos al dominio, esto con el fin de que el código sea más legible para el cliente y pueda ver las funcionalidades de dominio claramente. También se usa una arquitectura hexagonal o arquitectura de puertos y adaptadores para así lograr un desacoplamiento completo entre cada módulo del sistema, esto hace que la evolución de la aplicación sea más limpia y adaptable a nuevas funcionalidades.

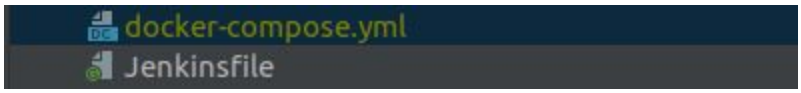
TESTING

Se desarrollaron pruebas unitarias para todos los métodos de todos los módulos de este servicio, esto con el fin de:

- mantener una cobertura del código mayor al 90 %
- Hacer el código más confiable
- usar prácticas de Integración continua / despliegue continuo del código.
- Asegurar el correcto funcionamiento del código del servicio.

DESPLIEGUE

Se usa Jenkins como servidor de automatización y Docker para la automatización de los despliegues del servicio, esto con el fin de automatizar todo el proceso de despliegue del servicio cuando sea necesario. Finalmente esta imagen de docker se despliega en una máquina de AWS haciendo uso del servicio de EC2.



LOGUEO

Se usa **LOG4J** como herramienta de mensajes de registro, esto a su facilidad de uso y compatibilidad con Java.

```
@Slf4j
```

RESULTADOS

Como resultado se obtuvo un microservicio modularizado como un Maven's Multi-Module Project, desarrollado en Java 11 con SpringBoot. Este proyecto inicia un servidor SIP en un puerto específico y al momento de recibir un SIP INVITE responde con dos puertos, los cuales habilita para recibir paquetes de audio. Posteriormente envía el audio recibido por estos puertos UDP al dominio, el cual transforma el audio de Mono Ulaw a formato WAV y lo envía a un servicio que contiene modelos estadísticos para determinar si el audio enviado corresponde o no a SPAM y/o audio malicioso. Finalmente, cuando recibe respuesta de los modelos, envía esta respuesta al ente encargado de realizar acciones sobre esta llamada en proceso.

CONCLUSIONES

- Los protocolos SIP/RTP a pesar de su antigüedad, siguen siendo una muy buena solución para VOIP
- SpringBoot facilita en gran manera muchos aspectos, entre ellos, la inyección de dependencias, lo cual ayuda a que sea aún más legible el código
- DDD como marco de trabajo conlleva muchas ventajas, entre ellas la legibilidad del código tanto para el cliente como para los desarrolladores
- La programación funcional como paradigma de programación facilita en gran manera los test unitarios y lectura del código.

REFERENCIAS

[1] Amazon, E. C. (2014). Amazon.
[2] Schwaber, K., & Beedle, M. (2002). Agile software development with Scrum (Vol. 1). Upper Saddle River: Prentice Hall.
[3] Holmström, Petter. (2019). Domain-Driven Design and the Hexagonal Architecture
[4] Jenkins. https://www.jenkins.io/
[5] Docker. https://www.docker.com/