ESCUELA COLOMBIANA DE INGENIERA JULIO GARAVITO

Plataforma escalable para el procesamiento de señales basada en un modelo de actores y canales reactivos

by

Luis Felipe Díaz Chica y Camilo Andres White

Programa de ingeniera de sistemas

24 de enero de 2017

ESCUELA COLOMBIANA DE INGENIERA JULIO GARAVITO

Resumen

Programa de ingeniera de sistemas

by Luis Felipe Díaz Chica y Camilo Andres White

Este proyecto es la continuación del proyecto Ïmplementación y análisis comparativo de una arquitectura NBED- IO (Non-blocking Event- Driven I/O) para una plataforma de telemetría.”buscando como valor agregado incluir las tecnologías de canales reactivos y escalabilidad para medir el rendimiento de transmisiones de señales cardiacas en tiempo real.

Índice general

Abstract II

List of Figures IV

List of Tables V

Abbreviations VI

1. Desarrollo del proyecto 1
- 1.1. Contexto . . . 1
  - 1.1.1. Planteamiento del problema y justificación . . . 1
  - 1.1.2. Marco teórico y estado del arte . . . 2
  - 1.1.3. Objetivos generales y específicos . . . 4

2. Arquitectura y tecnologías 6
- 2.1. Pruebas de concepto . . . 6
  - 2.1.1. Investigación de tecnologías . . . 6
- 2.2. Planteamiento de la arquitectura . . . 9
- 2.3. Análisis de la plataforma y el proyecto . . . 13

3. Pruebas y experimentos 17
- 3.1. Pruebas de carga . . . 17

4. contenedores 22
- 4.1. Docker . . . 22

5. Conclusiones 23

Bibliografía 24

Índice de figuras

2.1. Plataforma no bloqueante 7
2.2. Modelo de comunicación usando Backpressure 8
2.3. Demanda de datos por el subscriptor al procesador que tiene datos almacenados 8
2.4. Representación de actores remotos distribuidos 8
2.5. Arquitectura planteada 9
2.6. Arquitectura planteada 10
2.7. Definición del sistema de actores remoto transformadores 11
2.8. Archivo de configuración para los actores remotos que usa el servidor . . . 11
2.9. Interfaz implementada por los filtros que manipulan los datos de la señal . 12
2.10. Enrutamiento de mensajes cuando se recibe un dato 12
2.11. Código de AngularJs del publisher Web 13
2.12. Código de AngularJS del cliente observador 14
2.13. Diagrama de componentes del proyecto M-Health 15
2.14. Diagrama de secuencia del proceso de envío, transformación y comunicación de resultados de la señal. 16
2.15. Estructura de proyecto SBT dentro de IntelliJ 16

3.1. Configuración de conexiones con Gatling 18
3.2. configuración de acciones con Gatling 18
3.3. configuración de escenario con Gatling 18
3.4. configuración de acciones con Gatling 19
3.5. Resultados Escenario 1 20
3.6. Resultados Escenario 2 21

Índice de cuadros

3.1. Tabla de resultados escenario 1 20
3.2. Tabla de resultados del escenario 2 21

Abbreviations

ECI **E**scuela **A**Colombiana de **I**ngeniería Julio Garavito

Capítulo 1

Desarrollo del proyecto

1.1. Contexto

1.1.1. Planteamiento del problema y justificación

Actualmente el flujo constante de datos entre un cliente y un servidor ha tenido grandes dificultades en sus implementaciones con las tecnologías y arquitecturas tradicionales, se han presentado problemas como:

- Desaprovechamiento de recursos de cómputo
- Ahogamiento del cliente o el servidor ya que no hay una sincronización entre la cantidad de datos recibidos y la cantidad de datos enviados
- sobrepoblación de hilos corriendo en un servidor para resolver las peticiones que llegan constantemente.

Es necesaria una solución desde el punto de desarrollo y arquitectura de software que libere todos los problemas que actualmente se presentan.

Este proyecto de investigación tiene como proyección, a largo plazo, el desarrollo de una aplicación centralizada de procesamiento de señales biomédicas. Teniendo en cuenta que la misma tendrá que eventualmente recibir y procesar señales transmitidas por centenares de pacientes simultáneamente se requiere, además de una configuración de infraestructura adecuada, la evaluación de nuevos modelos de procesamiento de peticiones

concurrentes que podrían ayudar a alcanzar el desempeño esperado. En el proyecto de grado anterior "Implementación y análisis comparativo de una arquitectura NBED-IO (Non-blocking Event-Driven I/O) [1] para una plataforma de telemetría" se planteó una arquitectura basada en un modelo de actores, la cual, teóricamente, permitiría escalar la capacidad de la plataforma mediante el uso de actores remotos. Para el presente proyecto se retomará este trabajo, y se hará énfasis en la incorporación y pruebas del mecanismo de escalabilidad planteado, además de agregar un componente de procesamiento de datos usando canales reactivo que apunta a mejorar el desempeño de transmisión de datos en la arquitectura planteada.

1.1.2. Marco teórico y estado del arte

Los avances tecnológicos han estado ligados en paralelo a los avances en la medicina y actualmente se ha desarrollado el concepto de eHealth un concepto relativamente nuevo adoptado desde 1999 [2], consiste en el cuidado de la salud soportado por procesos electrónicos y tecnologías de la información y comunicación; entre las diferentes formas que existen de realizar procedimientos de eHealth está la Telemedicina y el mHealth. La Telemedicina consiste en brindar servicios de atención médica a distancia a una población de pacientes y ha sido tema de estudio para muchos sectores de la medicina, como las citas médicas remotas, atención a emergencias y una muy importante, el monitoreo remoto que ha tenido un alto crecimiento en los últimos años debido a la necesidad de incursionar en este modelo de servicio médico, según un experimento de Telemedicina realizado en pacientes con riesgos cardiovasculares en el Hospital María de Puerto Leguínazo en Putumayo arrojó que la Telemedicina [3] contribuye a un buen desarrollo de la práctica médica y ayuda a la rehabilitación. De la mano a todo lo anterior está el tema de mHealth, es una abreviación de mobile health que consiste en aplicar la medicina soportada por tecnologías móviles usando dispositivos de comunicación móviles, tabletas, PDAs y todas las tecnologías en temas de comunicación entre estos dispositivos.

Ejemplos de mHealth hoy encontramos muchos, en la universidad de Toronto Canadá se impulsó un proyecto 4 que ayudara a el control de la diabetes en jóvenes por medio de una aplicación móvil que tome registro de sus rutinas alimenticias diarias y estas eran monitoreadas por un médico experto en el tema, el proyecto arrojó que es un gran incentivo este nuevo modelo de servicios de control médico [4].

Estas nuevas áreas de investigación en la medicina han ido evolucionando de la mano con los avances de la tecnología, y actualmente encontramos que la mayoría de las soluciones que implican monitoreos en tiempo real relacionado con Telemedicina y mHealth están basados en arquitecturas donde la eficiencia de transmisión de los datos no es la mejor, y no es culpa de los desarrolladores, ni los arquitectos, es culpa de la naturales limitaciones que los avances en la ingeniería electrónica y en la ingeniería se software imponen. Hasta el momento se han desarrollado soluciones muy interesantes en áreas de monitoreo ECG [5] , usando las tecnologías que el mundo ofrece, pero hay muchas deficiencias en temas de concurrencia, el modelo actual de concurrencia implica crear hilos por cada petición al servidor, lo que genera una lucha por el procesador y de esta forma, la transmisión de datos en tiempo real desde varios clientes a un mismo servidor se convierte en un pico de botella perpetuo . Es de total interés analizar las nuevas ideas que investigadores e interesados han propuesto desde el punto de vista del desarrollo de software para dar solución a los problemas que la tecnología actual trae consigo, una iniciativa que ha tomado fuerza son los canales reactivos (reactive streams), ha sido un nuevo modelo planeado para atacar los problemas que se presentan en temas de transmisión de datos concurrentes en tiempo real, un problema bastante específico que sucede no sólo en las áreas de la medicina, también en servicios de transmisión multimedia en tiempo real (Streaming) [6] , con el fin de mejorar la eficiencia de transmisión datos.

Hoy en día, la gran cantidad de información que se transporta en la red exige a los sistemas estar en constante alerta y con alta disponibilidad, mantener esa gran cantidad de información fluyendo ha requerido una reestructuración en la forma de transmitir la información, por lo cual gracias a los emprendimientos de empresas como: Typesafe, Netflix, Pivotal, Oracle, han establecido que la mejor manera de hacerlo es a través de canales reactivos, donde la información fluye a un ritmo constante y sin fin, donde la velocidad de transmisión de datos ya no depende de los publicadores o suscriptores, sino de un sistema altamente escalable y con tiempos de respuesta bastante cortos. Las primeras discusiones ocurrieron en el 2013 entre los equipos de Akka y Play [7]; aunque para entonces el streaming aún era complejo de implementar, de esta manera iniciativas como Akka, RxJava, Project Reactor, VertX, Slick los cuales son tecnologías basadas en: Java, Scala, JavaScript, Groovy, Ruby, Ceylon; llevan avanzado la investigación de diferentes modelos con diferentes lenguajes para la implementación de canales reactivos.

Algunas aproximaciones en modelos web y plataforma de computación en la nube han

planteado modelo de monitoreo en tiempo real usando las tecnologías de la computación en la nube.La arquitectura es casi la misma en todas las soluciones de este estilo, existe un servidor médico en la nube y un dispositivo móvil que actúa como cliente junto al paciente. El cliente envía mensajes a un entorno web donde un software de visualización es necesario para ver los paquetes recibidos y se han usado diferentes tecnologías para lograr esta arquitectura entre ellas, JSP (Java Server Pages), Php; la mayoría usando un método de transmisión de datos por paquetes [8].

El proyecto anterior buscaba proponer una arquitectura que diera un giro a las arquitecturas convencionales cliente-servidor sobre hilos por petición, y se concluye que una arquitectura de software basada en un modelo de actores no bloqueantes debería ser considerada en proyectos de Telemetría, ya que el flujo de grandes cantidades de información demanda un uso eficiente de los recursos de cómputo. Esta aproximación se apoya en el modelo de programación asíncrona, que buscar hacer un uso eficiente de los recursos de la máquina sin generar picos de botellas en la utilización de la unidad de procesamiento. Es objeto de este proyecto atacar el mismo problema investigando si una arquitectura basada en canales reactivos y actores con posibilidad de escalabilidad tiene mejores resultados en eficiencia del uso de capacidad de cómputo.

1.1.3. Objetivos generales y específicos

Objetivo general:

Implementar una plataforma para el procesamiento y análisis de señales biométricas basada en un modelo de actores y canales reactivos y el principio de Non-blocking Event-Driven I/O.

Objetivos específicos:

- Dar continuidad al proyecto “Implementación y análisis comparativo de una arquitectura NBED-IO(Non-Blocking Event-Driven I/O) para una plataforma de telemetría”, en el cual se desarrolló una arquitectura basada en un modelo de actores para el procesamiento de señales.

- Desarrollar y evaluar diferentes esquemas de “despacho” de actores remotos, el cual permita satisfacer los requerimientos de escalabilidad de una plataforma de procesamiento de señales.
- Redefinir y reimplementar el protocolo de comunicaciones para la transferencia de señales entre el sensor y la plataforma, de manera que se reduzcan los tiempos de latencia, la carga sobre la res y por ende, se mejore la eficiencia energética(back pressure).
- Incorporar a la arquitectura un mecanismo de persistencia para las señales que sea compatible con el principio de eventos no-bloqueantes.
- Migrar nuevos algoritmos de procesamientos de señales a la plataforma, y realizar pruebas de carga sobre los mismos. Implementar un mecanismo de transmisión constante de datos sobre canales reactivos.

Capítulo 2

Arquitectura y tecnologías

2.1. Pruebas de concepto

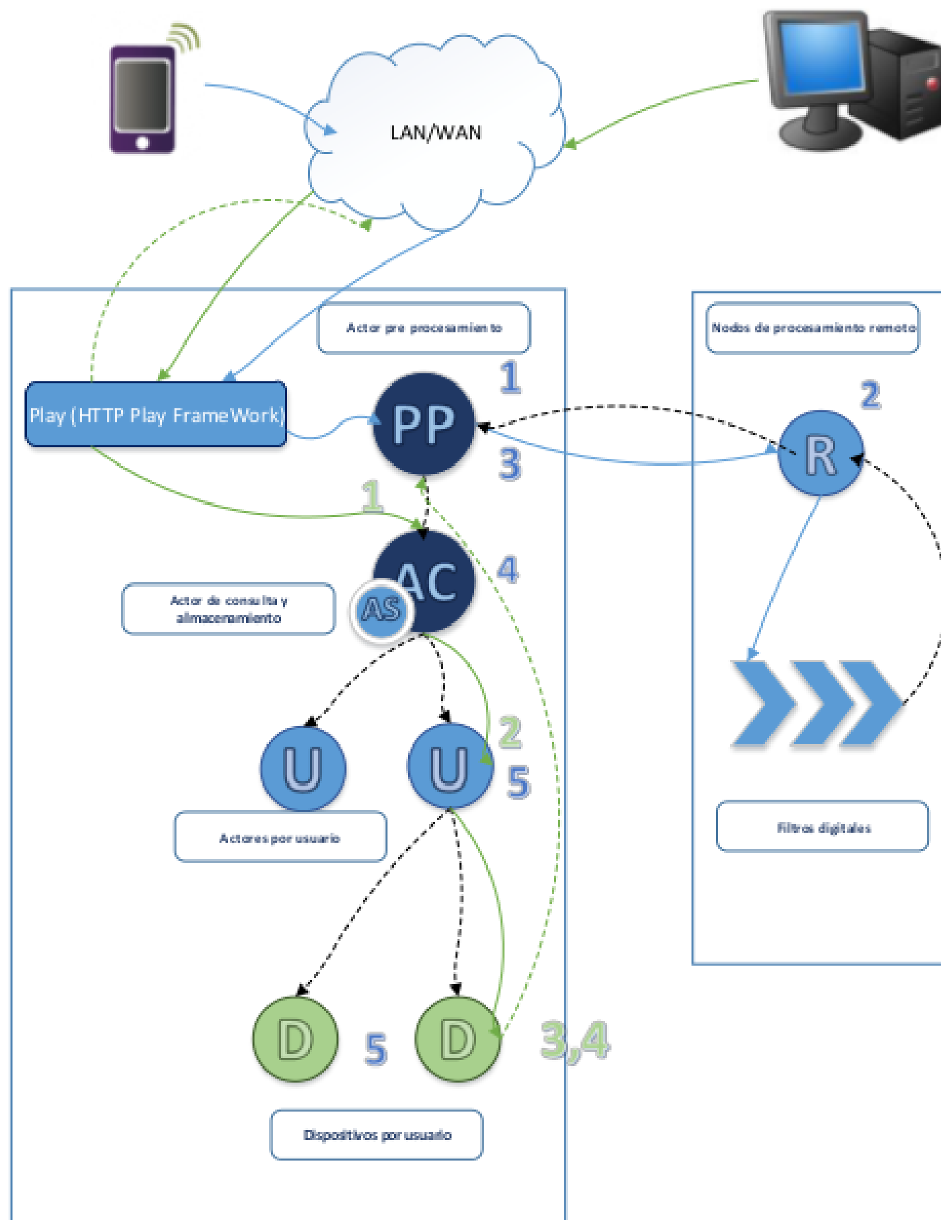
2.1.1. Investigación de tecnologías

Desde el proyecto anterior se realizaron las comparativas entre tecnologías que podrían solucionar la problemática del proyecto en general, y se llegó a la conclusión usar el lenguaje funcional Scala y su framework Akka para brindar una plataforma No bloqueante de la mano con el modelo de actores que tiene implementado Scala. En esta primera instancia del proyecto se propuso una arquitectura que usaba canales de comunicación convencionales, usando un framework de API REST llamado PLAY que permite la comunicación bajo este estándar usando el protocolo Http [1].

Tomando como punto de partida los diseños del proyecto anterior y los objetivos del nuevo proyecto, se comenzó la fase investigativa que apuntaba a cumplir con estos nuevos objetivos tecnológicos.

El primero de estos estuvo orientado a entender un sistema reactivo y así mismo cómo este nuevo modelo puede aplicarse a este proyecto. se encontró que a pesar de ser un paradigma nuevo, existen ya muchas compañías que han implementado librerías de canales reactivos. Entre las estudiadas encontramos las librerías de Akka, RxJava, Slick, Vert.x, Project Reactor.Entre todas estas se decidío por usar Akka con su librearía akka-streams [9], esto por lo que el proyecto anterior había usado componentes de la misma librería

FIGURA 2.1: Plataforma no bloqueante



para la escalabilidad y era una oportunidad para seguir construyendo el proyecto con las mismas tecnologías y tomar las ventajas de la programación funcional en Scala.

Al usar los canales reactivos estamos aprovechando la capacidad de hacer back-pressure Fig. 2.2, este es un nuevo mecanismo de como llevar datos de un punto a otro de forma asincrónica [6] utilizando la máxima capacidad de rendimiento y recursos entre los actores de la comunicación Fig. 2.3.

Por otro lado el tema de los actores remotos que ya se había estudiado en el proyecto

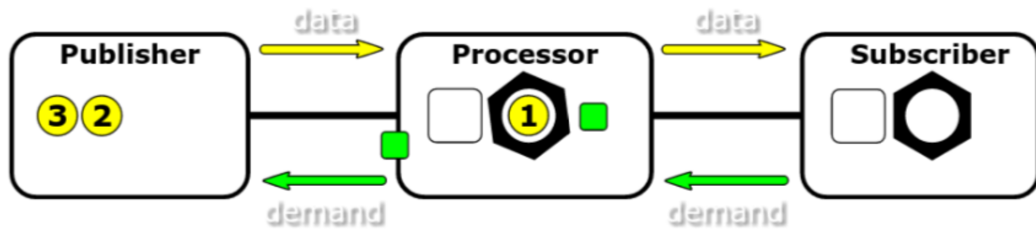


Figura 2.2: Modelo de comunicación usando Backpressure

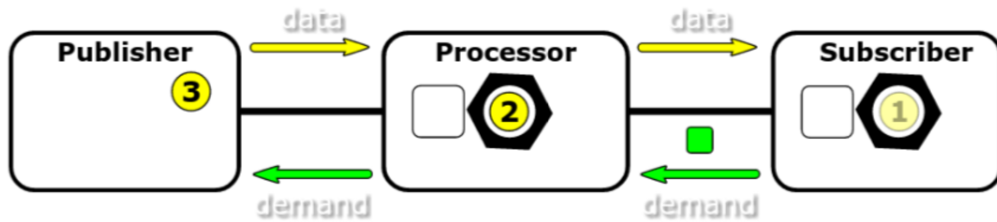


Figura 2.3: Demanda de datos por el subscriptor al procesador que tiene datos almacenados

anterior, se sigue manteniendo para usar esta tecnología en conjunto con lo nuevo que se quiere agregar al proyecto. Se usará la librería de akka-remote en Scala.

Se realizaron varias pruebas de concepto sobre akka para entender el funcionamiento de los actores remotos y los canales reactivos, como primera instancia se probaron los conceptos de actores remotos, ya que akka-remote permite crear y buscar actores que están distribuidos a través de una red. Fig.2.5

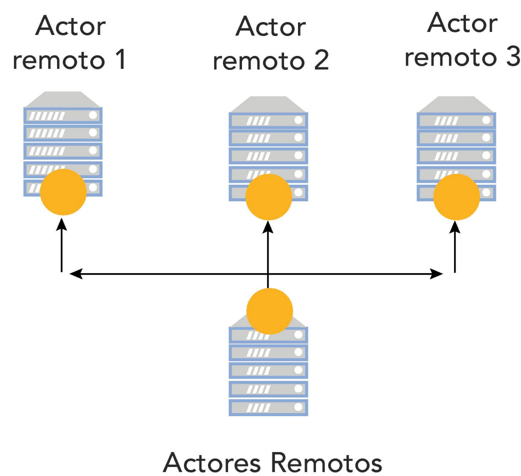


Figura 2.4: Representación de actores remotos distribuidos

Luego de esto entendimos como por medio de canales reactivos se puede crear una puerta de comunicación a través de WebSockets, de esta forma es posible generar un punto de entrada de peticiones a un servidor y estas peticiones al ser atrapadas por el WebSocket, son convertidas en flujos usando la librearía de akka-streams [10].

2.2. Planteamiento de la arquitectura

Después del análisis realizado se planteó la siguiente arquitectura, tomando como referencia las pruebas de concepto, el proyecto anterior y algunas cuántas ideas que se fueron puliendo.

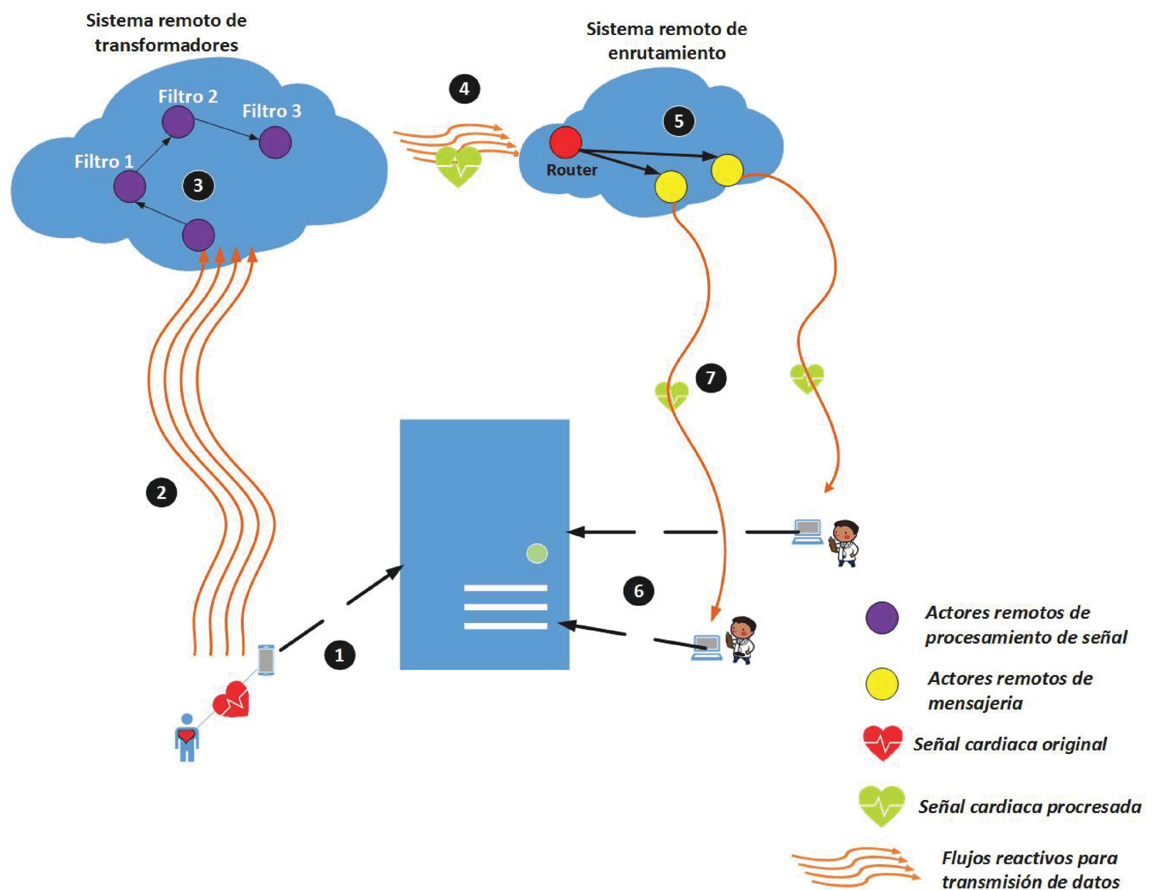


FIGURA 2.5: Arquitectura planteada

Para ir más al detalle se va a explicar cada parte de la plataforma.

Comenzamos por el punto (1). Para crear un canal de comunicación desde el cual los pacientes pudieran enviar datos, se combinó el protocolo de comunicación WebSockets junto con los canales reactivos, esto nos permitió brindar un API de conexión y envío de datos accesible a cualquier externo, ya sea desde un browser, un medidor de señales médicas conectado a internet, un celular, etc. lo importante era que se siguiera la especificación. Una vez abierto el WebSocket este era conectado a un flujo reactivo creado con akka-stream, aquí entra a jugar la inteligencia del backpressure, ya que si el paciente enviaba muchos datos al tiempo, el canal iba a manejar los datos a su ritmo, sin ser ahogado.

```
val service = {
  path("publish" / IntNumber) { id =>
    get {

      handleWebSocketMessages(FlowResponses.publish(transformer, id))
    }
  } ~
  path("subscribe" / IntNumber) { id =>
    get {

      handleWebSocketMessages(FlowResponses.subscribeToFlow(router, id))
    }
  }
}
val bindingFuture = Http().bindAndHandle(service, interface, port)
```

FIGURA 2.6: Arquitectura planteada

Se crearon dos puntos de acceso, uno era el punto "Publish"(Fig 2.6) , desde allí se conecta cualquier entidad que quiera transmitir una señal, sólo es necesario agregar un identificador a la señal como parámetro del URI, y el segundo punto de acceso es para los observadores, el "Subscribe"(Fig 2.6). desde allí cualquier tercero podría subscribirse a una señal que estuviera enviado un publicante, la una condición era enviar tanto en el Publisher como en el Subscriber el identificador de la señal el cuestión. Aquí se hizo uso de un framework para Akka llamado Spray, este nos permite escribir de forma sencilla las servicios web que ofrece el servidor usando la sentencia path() junto con el método del request como se puede ser ver en la figura 2.6.

Un punto a resaltar es como el Request es transformado a WebSocket y se liga con un flujo de akka, todo eso sucede con el método handleWebSocketMessages() de Spray [11], de esta forma los servicios que va a ofrecer el servidor son pasados a el método Http()

que de forma muy sencilla permite levantar el servidor sobre un puerto y una direcciòn (interfaz)

```
val transformer: ActorSelection = serverSystem.actorSelection(configFile.getString("server.actors.transformer"))
```

FIGURA 2.7: Definición del sistema de actores remoto transformadores

En el punto (2): Se ha establecido una conexión entre el flujo que está atendiendo el envío de datos del publicante con un sistema de actores remoto, al cual llamamos ***Sistema Remoto de Transformadores***, como se puede ver en la definición del WebSocket, al punto de acceso Publish se le está pasando como parámetro este sistema de actores externo que se define con las especificaciones de akka-remote Fig 2.7.

En el punto (3): La conexión del servidor principal con el sistema remoto de transformadores está definida en un archivo de configuración que akka-remote usa para encontrar exactamente en donde se encuentra el sistema de actores. Como se ve en la imagen 2.8, el sistema remoto de transformadores para esta configuración está ubicado en la dirección IP 10.2.67.92 en el puerto 3030, se esta forma akka sabe como crear el canal de comunicación.

```
server {
  interface = "127.0.0.1"
  port = 9001
  system {
    name = "server"
  }
  actors {
    router = "akka.tcp://Router@10.2.64.90:2020/user/router"
    transformer = "akka.tcp://Transformers@10.2.67.92:3030/user/transformerActor"
  }
  methods {
    publish = "/publish"
    publish = "/subscribe"
    error = "Invalid websocket request"
  }
  online = "Server online at http://"
  error = "Server took to long to startup, shutting down"
}
```

FIGURA 2.8: Archivo de configuración para los actores remotos que usa el servidor

En este punto todos los datos que son enviados por el publicante pasan por una serie de transformaciones que son procesadas por actores que tienen esta responsabilidad, para esta situación se han encadenado en forma serializada las transformaciones Fig. 2.9. Estos actores han implementado un sistema de persistencia para recordar datos anteriores de una señal en particular, esto con el fin de usar esta memoria en los algoritmos de

transformación que requieren calcular un valor con base a valores del pasado, este es el caso para el algoritmo de suavizado que fue implementado en este caso.

```
trait Transformer extends Actor with Node {

  val memory = new LocalMemory

  override def receive = {
    case TransformSignal(signal: Signal) => {
        nextNode ! TransformSignal(transformSignal(signal))
    }
    case RegisterNextTransformer(tr: ActorRef) => {
      setNextNode(tr)
    }
  }

  def transformSignal(signal : Signal) : Signal
}
```

FIGURA 2.9: Interfaz implementada por los filtros que manipulan los datos de la señal

En el punto (4) sucede de la misma forma como el servidor principal registró el sistema de actores, pero esta vez el sistema remoto de transformadores registra un ***Sistema Remoto de Enrutamiento***, este se encargará de enrutar los resultados que arroje el proceso de transformación a todos los clientes que estén subscritos a una señal, el enrutador alberga un actor por cliente que se haya , de esta maner el fallo de un actor es transparente para las demás conexiones abiertas, una de las muchas ventajas que tiene el sistema de actores remotos, es el manejo de errores y recuperación de los mismos.

En el punto (5) tenemos el sistema remoto de enrutamiento, en esta hay un enrutador que conoce todos los actores que tienen una conexión con algún subscriber, de esta forma cuando recibe un mensaje del sistema remoto de transformadores, inmediatamente procede a enrutar el mensaje a los actores que les interesa, eso lo hace usando el identificador de la señal que trae dato Fig.2.10.

```
override def receive = {
  case msg : Signal => routees
      .filter(i => i._2 == msg.getId)
      .foreach(i => i._1.send(msg.getValue.toString, sender))
```

FIGURA 2.10: Enrutamiento de mensajes cuando se recibe un dato

En el punto (6) sucede algo muy parecido a lo que sucedía en el punto (1), en este caso el canal es abierto sobre el socket subscribe, pero nunca se están enviado datos, para esta caso se considera una entidad observadora que espera que el WebSocket envíe los datos, ya es independiente de cada tercero como interprete los datos.

El punto (7) termina siendo el envío y recepción de los resultados de la transformación a los interesados, en nuestro caso podrían ser doctores, familiares o cualquier persona que quisiera monitorear en tiempo real el estado del paciente.

```
$scope.sendStreaming = function () {
    readFile();
    $scope.closeSocket();
    ws = new WebSocket("ws://10.2.67.90:9001/publish/1");
    ws.onopen = onOpen;
    ws.onclose = onClose;

    if(ws != null) {
        interval = $interval(function () {

            if(counter == valores.length) counter = 0;
            var newValue = valores[counter];
            counter++;
            chart.data.datasets[0].data.push(newValue);
            chart.data.datasets[0].data.shift();
            chart.update();
            ws.send("" + newValue);
        }, 1000);
    }
};
```

Figura 2.11: Código de AngularJs del publisher Web

Para el envío de datos en el paso (1) hemos utilizado un cliente web construido sobre AngularJS, este cliente toma datos medidos de una señal cardiaca real, son graficados en el browser y enviado al servidor Fig. 2.11.

Para el caso del observador hemos usado un cliente Web construido con AngularJS que se conecta al WebSocket, recibe los datos y los muestra en una gráfica usando la librería ChartJS 2.12.

2.3. Análisis de la plataforma y el proyecto

Un diagrama de componentes de la plataforma sería el que se ve en la figura Fig 2.13, allí se puede ver que hay tres componentes importantes, el gateway, transformer y el router, también se han diagramado los servicios que cada componente ofrece.

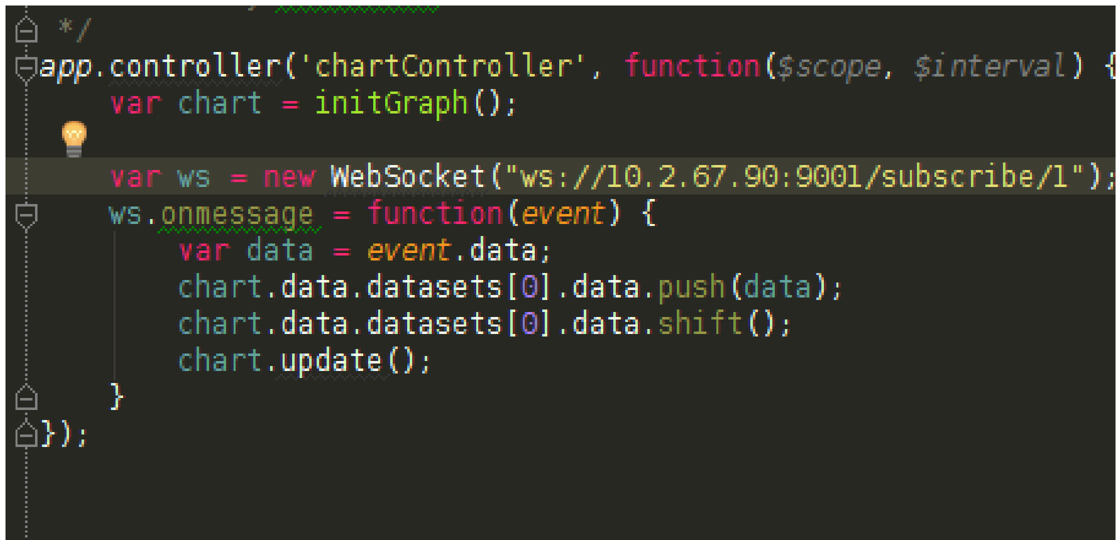


FIGURA 2.12: Código de AngularJS del cliente observador

En el diagrama de secuencia se puede evidencias como es la responsabilidad que cada componente está adquiriendo en todo el proceso de envío, transformación y comunicación de la señal. Se puede detallar que en el componente del transformador existen unos nodos dedicados a l transformación de la señal, lo mismo sucede con el componente del router, allí existen nodos por cada conexión que es abierta por un subscriptor, estos nodos son dedicados a la conexión y conocen la información necesaria para dar una respuesta a los stakeholders de la señal.

El proyecto a nivel de configuración ha quedado ligado a un sólo proyecto en IntelliJ llamado App, este es un proyecto de naturaleza SBT, y dentro de este proyecto se encuentran los demás módulos que hacen parte del proyecto, entre ellos los tres componentes principales mas dos adicionales que no han sido mencionados, ya que su funcionalidad es más dar utilidad al proyecto que crear nuevas funcionalidades. Uno de estos es el módulo çommon"2.15, en este encontramos todos los artefactos del proyecto que son comunes para todos los módulos, como por ejemplo la clase señal, es una clase general para todo el proyecto y es por esto que nace la necesidad de crear un módulo que centralice eso.

Otro módulo que fue necesario crear fue el módulo de persistencia, este surge por la necesidad de que los actores de los tres componentes principales pudieran recordar información. Por naturaleza del modelo de actores, no existe memoria en los actores, ya que su función principal es enviar y recibir información y actuar según el mensaje recibido. Usando una estructura de datos particular, se pudo lograr almacenar información dentro de los actores, específicamente puntos de la señal pasando en el tiempo.

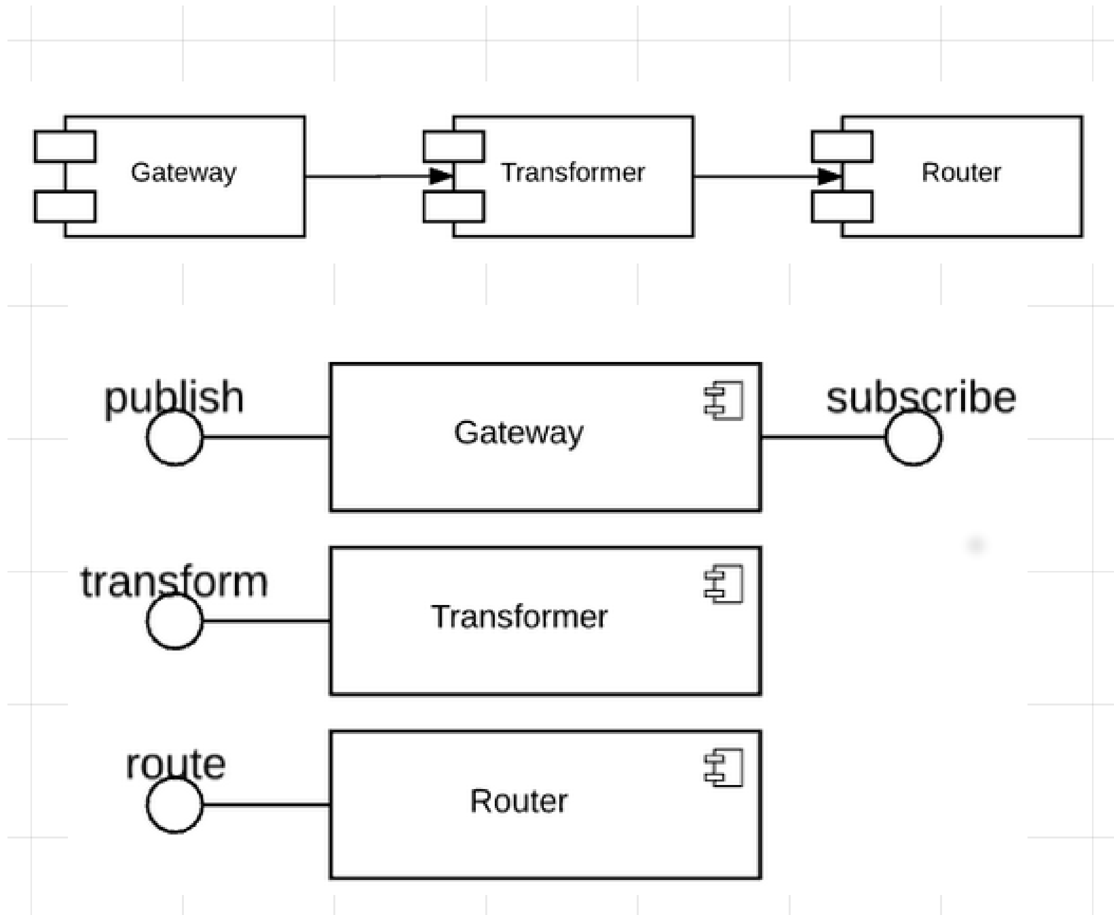


Figura 2.13: Diagrama de componentes del proyecto M-Health

La plataforma propuesta le apunta a mejorar el desempeño en tiempos de respuesta al enviar datos, procesarlos y dar una respuesta. Con esto estamos listos para realizar pruebas y poder determinar si la plataforma arroja resultados positivos.

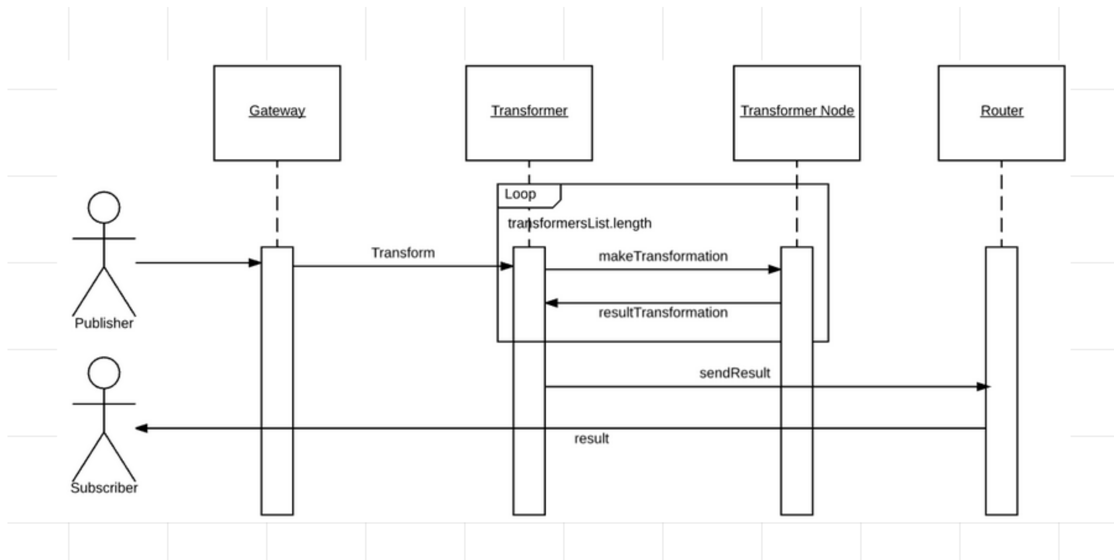


FIGURA 2.14: Diagrama de secuencia del proceso de envío, transformación y comunicación de resultados de la señal.

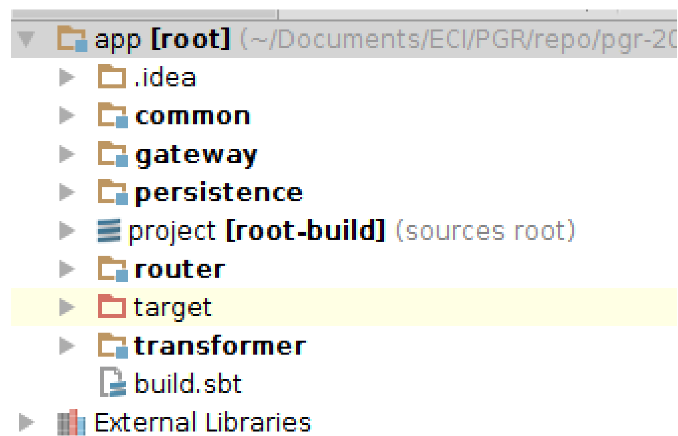


FIGURA 2.15: Estructura de proyecto SBT dentro de IntelliJ

Capítulo 3

Pruebas y experimentos

3.1. Pruebas de carga

En la fase de pruebas, se trabajo en la creación de escenarios en los que se pudiera probar la plataforma propuesta VS una plataforma convencional que no usa las tecnologías que se han venido investigando a lo largo del proyecto, para esto se investigaron dos herramientas creadas especialmente para realizar pruebas de carga y estrés sobre protocolos de comunicación HTTP y WebSockets, las herramientas contempladas fueron Gatling y Artillery, Gatling ofrece la posibilidad de programar escenarios en lenguajes Java y Scala, por otro lado Artillery maneja archivos de configuración JSON y XML. Por temas acoplamiento con el proyecto se decidió usar Gatling ya que se puede manejar Scala y hace que el proyecto sea más uniforme en los lenguajes de programación que lo componen y aprovechar más lo aprendido durante la curva de aprendizaje de todo el proyecto.

Las pruebas de gatling se conocen como simulaciones y usan una estructura que se describe a continuación.

Un encabezado donde se configura el tipo de conexión que se va a probar, puede ser Http o WebSockets (ver Imagen 3.1)

Luego se esto de configuran las acciones que va a realizar cada señal o usuario , los pasos se describen en una sintaxis de programación muy entendible. Lo primero que se hace es

```
val wsTestConf = http
  .baseURL("http://localhost:80")
  .acceptHeader("text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8")
  .doNotTrackHeader("1")
  .acceptLanguageHeader("en-US,en;q=0.5")
  .acceptEncodingHeader("gzip, deflate")
  .userAgentHeader("Gatling2")
  .wsBaseURL("ws://127.0.0.1:9001")
```

FIGURA 3.1: Configuración de conexiones con Gatling

tomar los datos que se van a enviar de un archivo ”signalsdata.csv”, a esto se le conoce como *feeder*, luego el usuario se subscribe al canal y comienza a enviar todos los datos que están dentro del archivo, cada vez que un dato es enviado, el usuario igualmente se subscribe al canal de recepción de datos y espera un tiempo máximo de 60 segundos por una respuesta en ese canal (ver imagen 3.2).

```
object WsSend {
  val feeder = csv("signaldata.csv").records // default is queue, so for this test, we use random to a
  exec(session => session.set("id", session.userId))

    .exec(ws("Connect WSS").wsName("set").open("/publish/${id}"))
    .exec(ws("Connect WSG").wsName("get").open("/subscribe/${id}"))
    .foreach(feeder, "context"){
      exec(flattenMapIntoAttributes("${context}")) //non-shared data between the feeders
        .exec(ws("Send msg ws").wsName("set").sendText("${data}"))
        .exec(ws("Receive WSG").wsName("get").check(wsListen.within(60 seconds).until(1).regex(".*")))

    }.exec(ws("Close WSS").close).exec(ws("Connect WSG").close)
```

FIGURA 3.2: configuración de acciones con Gatling

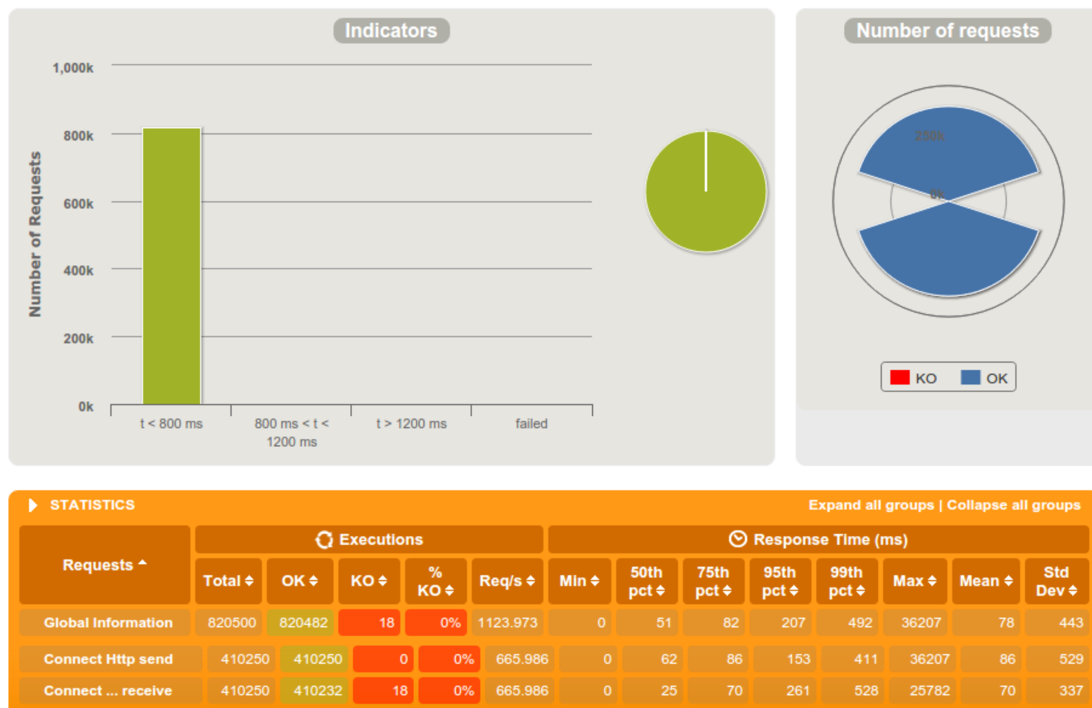
Por ultimo hay que configurar el escenario, allí se le indica a gatling cuántos usuarios se van a lanzar y qué acciones tomaran (Imagen 3.3).

```
val wsSendScenario = scenario("WebSocket send communitacion").exec(WsSend.send)
setUp( wsSendScenario.inject(atOnceUsers(100)).protocols(wsConf))
```

FIGURA 3.3: configuración de escenario con Gatling

Gatling genera un reporte en html con el resultado del escenario detallando cada punto de las acciones que se inyectaron en el escenario, en el casio de la imagen, son los resultados para el lanzamiento en http (Imagen 3.4).

Escenarios



Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	820500	820482	18	0%	1123.973	0	51	82	207	492	36207	78	443
Connect Http send	410250	410250	0	0%	665.986	0	62	86	153	411	36207	86	529
Connect ... receive	410250	410232	18	0%	665.986	0	25	70	261	528	25782	70	337

FIGURA 3.4: configuración de acciones con Gatling

La plataforma propuesta la llamaremos *Plataforma MHealth*; dado que la plataforma esta dividida en tres componentes independientes, de los cuales, uno de estos se encarga de recibir y transmitir datos, por lo que basto con cambiar este componente para recibir las señales con distintos protocolos y que cumpla la misma funcionalidad, a esta alternativa le llamaremos: *Plataforma Convencional*.

La *Plataforma Convencional* fue construida bajo el modelo bloqueante usando servicios REST con el framework propuesto por **Play**, el cual ofrece una gran cantidad de utilidades para construir servicios REST, el cual tiene soporte tanto para **Scala** y **Java**.

Escenacio 1

Plataforma MHealth con canales reactivos distribuidos vs Plataforma MHealth sin canales reactivos usando servicios REST.

Para este escenario se hicieron 8 pruebas con gatling (ver Tabla 3.1), dentro de las pruebas se probaron varias señales concurrentes enviadas a las plataformas y se midieron los tiempos promedios de respuestas que Gatling arrojó.

Con estos resultados pudimos graficar las curvas y determinar que la plataforma tradicional, usando un modelo bloqueante tarda mucho más tiempo en responder que lo que tarda la plataforma MHealth en las condiciones del escenario 1 (ver Figura 3.5).

Cuadro 3.1: Tabla de resultados escenario 1

Plataforma MHealth		**Plataforma Tradicional**	
Señales concurrentes	**Tiempo promedio(ms)**	**Señales concurrentes**	**Tiempo promedio(ms)**
1	10	1	17
10	41	10	60
50	162	50	194
100	521	100	665

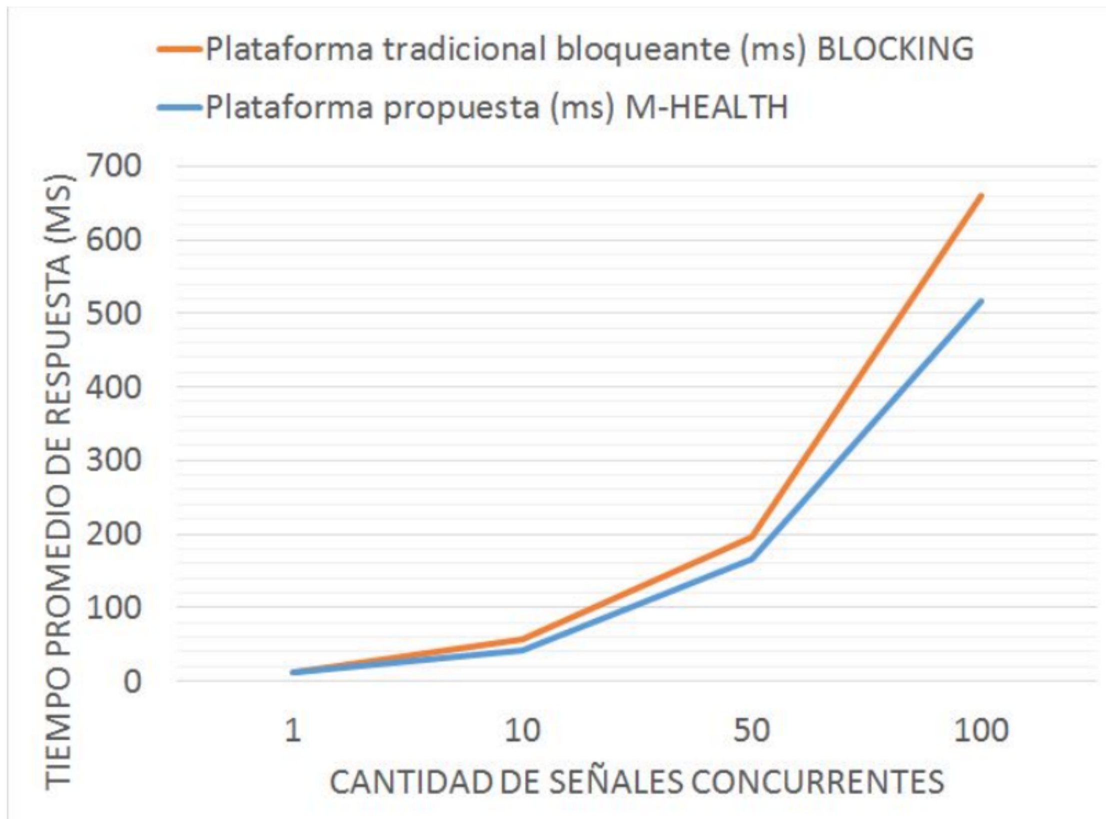


Figura 3.5: Resultados Escenario 1

Escenacio 2

Plataforma MHealth con canales reactivos distribuidos usando WebSockets vs Plataforma convencional sin canales reactivos usando WebSockets.

Para este escenario de preparó un ambiente distribuido donde la plataforma Mhealth hiciera uso del modelo de actores remotos y así mismo de los canales reactivos, se usaron exactamente 4 máquinas configuradas para todo el flujo para la *Plataforma MHealth*

Se realizaron 10 pruebas detalladas en el cuadra de abajo, aumentano del número de señales concurrentes para probar tiempos de respuesta.

Cuadro 3.2: Tabla de resultados del escenario 2

Plataforma MHealth		**Plataforma Tradicional**	
Señales concurrentes	**Tiempo promedio(ms)**	**Señales concurrentes**	**Tiempo promedio(ms)**
1	8	1	13
10	35	10	55
50	150	50	178
100	501	100	620
200	797	200	865
500	905	500	978

Fue evidente también en este escenario que hacer el procesamiento sigue siendo más rápido en la plataforma MHealth, queda claro decir que las conexiones realizadas entre las máquinas distribuidas es por medio de una red local, esto puede condicionar el experimento realizado ya que no existían barreras de ningún tipo en la comunicación entre las máquinas.

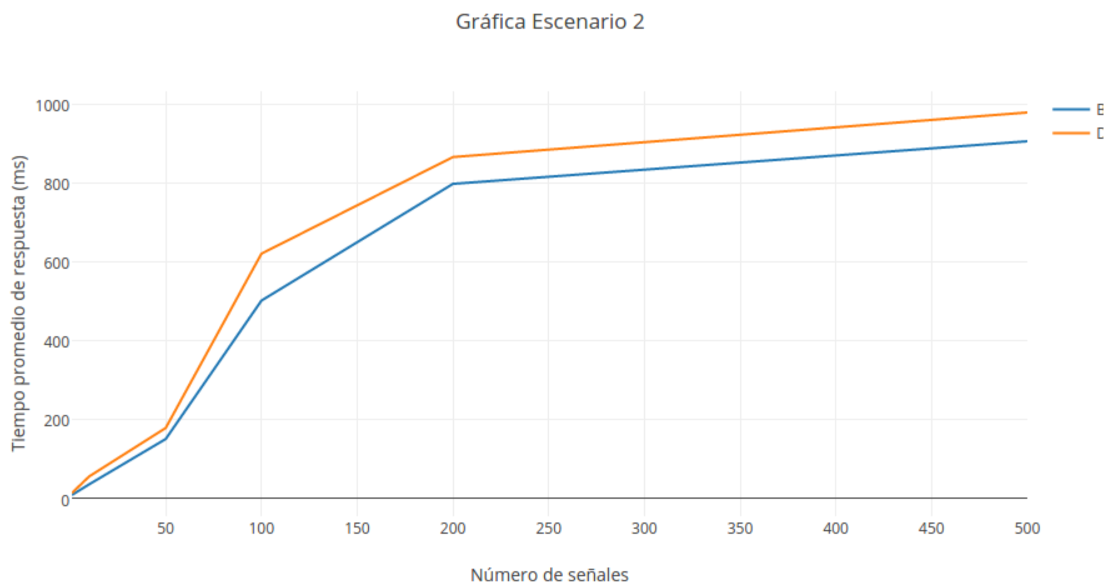


Figura 3.6: Resultados Escenario 2

Capítulo 4

contenedores

4.1. Docker

Docker fue la alternativa usada para lograr incrementar la infraestructura en caso de que sea requerido del proyecto. Esto lo debemos a que: al haber sido construida en diferentes componentes, es fácil desplegar múltiples estancias de cada componente para que estos interactúen sin necesidad de realizar re-despliegues de la aplicación. Esta forma de funcionamiento se debe gracias al auto descubrimiento de nuevos nodos a enlazar con la aplicación, de tal manera que cuando un nuevo nodo se adjunta, a partir de diferentes estrategias empezara a recibir y enviar mensajes.

De que manera funciona Docker entonces?. Docker es una plataforma que permite desplegar componentes de manera virtual izada usando los componentes que la maquina en la que se aloja ofrece. De esta manera los componente usados cuentan con instrucciones para obtener la ultima versión de la aplicación y ponerla en funcionamiento, es a esto a lo que nosotros llamamos como nodos, que como lo mencionamos antes, serán descubiertos y empezaran a funcionar junto a la aplicación.

Capítulo 5

Conclusiones

Durante el desarrollo e investigación del tema propuesto para el proyecto de grado, puntualmente: canales reactivos, nos enfrentamos a situaciones en las que nos dimos cuenta que los canales reactivos son una parte del todo que conforma una arquitectura realmente eficiente.

Los canales reactivos por si solos, facilitaran el evitar perdida de mensajes en el emisor y el receptor, evitaran la negación de algún servicio al controlar la cantidad de paquetes enviados, aumentaran la velocidad de transmisión de paquetes al usar canales dedicados; pero si la entidad responsable de procesar estos datos, no tiene la arquitectura adecuada, solamente estaremos creando un cuello de botella donde los datos no serán procesados y revisados posteriormente.

Los canales reactivos junto a una arquitectura reactiva/no bloqueante son realmente lo que conforma el todo de una aplicación capaz de cumplir con los pilares de la programación reactiva, la cual debe ser resilente, responsiva, elástica y orientada a mensajes; este tipo de aplicaciones son capaces de adaptarse en su totalidad a los diferentes tipo de carga, fallos e incluso nueva infraestructura, no solamente a la comunicación entre cliente y aplicación como lo mencionado antes.

De esta manera, luego de que los diferente experimentos lo demostraran, los diferentes resultados que obtuvimos usando canales no reactivos (REST), canales reactivos junto a WebSockets, incluida la aplicación reactiva y no reactiva encontramos que el valor aportado por la aplicaciones totalmente reactivas aumentan significativamente el rendimiento de las aplicaciones y que su valor lo demuestran los experimentos realizados.

Bibliografía

[1] M. A. Corredor Acosta and H. F. Cadavid Rengifo. M-health system backend supported by an actors model. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 150–156. doi: 10.1109/ColumbianCC.2015.7333409.

[2] Healthcare | free full-text | mobile tele-mental health: Increasing applications and a move to hybrid models of care | HTML. URL http://www.mdpi.com/2227-9032/2/2/220/htm.

[3] Telemedicina aplicada a la valoración del riesgo cardiovascular: Experiencia en el hospital maría angelines de puerto leguízamo, putumayo | osorio lópez | CIENCIA e INNOVACION EN SALUD. URL http://publicaciones.unisimonbolivar.edu.co/rdigital/ojs/index.php/innovacionsalud/article/view/536.

[4] JMIR-design of an mHealth app for the self-management of adolescent type 1 diabetes: A pilot study | cafazzo | journal of medical internet research. URL http://www.jmir.org/2012/3/e70/?trendmd-shared=1.

[5] Saurabh Prakash and V Venkatesh. Real time monitoring of ECG signal using PIC and web server. 5.

[6] Lightbend Inc. Async, back-pressure, immutability, resilience, concurrency and more from typesafe at devoxx - @lightbend. URL https://www.lightbend.com/blog/async-back-pressure-immutability-resilience-concurrency-and-more-from-typesafe-at

[7] Kevin Webber. A journey into reactive streams. URL https://medium.com/@kvnwbbr/a-journey-into-reactive-streams-5ee2a9cd7e29#.jc7ro3ic5.

[8] Remote ECG monitoring system. URL http://www.google.com/patents/US3986498. U.S. Classification 600/508, 128/904, 340/870.27, 340/870.39, 340/870.18, 340/636.15, 340/636.1, 128/903, 600/519; International Classification

A61B5/00; Cooperative Classification Y10S128/903, Y10S128/904, A61B5/0006; European Classification A61B5/00B3B.

[9] Akka streams documentation. URL http://doc.akka.io/docs/akka/2.4.14/scala/stream/index.html.

[10] Reactive streams with akka streams. URL http://www.infoq.com/news/2014/04/reactive-streams-akka-streams.

[11] Elegant, high-performance http for your akka actors. URL http://spray.io/.