# Plataforma escalable para elprocesamiento de señales basada en unmodelo de actores y canales reactivos

Luis Felipe Diaz, *Estudiante, ECI,* Camilo Andres White, *Estudiante, ECI,* Héctor Fabio Cadavid Rengifo, *Director de proyecto, ECI*

***Resumen*—El propósito sobre el cual se realizo la investigación de este proyecto fue la de demostrar la efectividad de los canales reactivos sobre medios de transmisión de datos convencionales como lo es REST, además de esto se realizaron diferentes experimentos con distintas arquitecturas (bloqueantes y no bloqueantes), aparte del uso de WebSockets para el uso de canales reactivos.**

***Keywords*—Canales reactivos, Arquitectura, Bloqueantes, REST, WebSockets.**

## I. Introducción

En la actualidad existe una gran cantidad de información siendo transmitida en tiempo real hacia millones de usuarios desde plataforma increíblemente complejas, esperando que estas sean capaces de entregar la información a sus respectivos usuarios en los tiempos que ellos esperan de al menos milisegundos en sistemas que operen 24/7. Diferentes soluciones y propuestas se han realizado, dentro de esta existen a las que se les denomina aplicaciones reactivas, las cuales empiezan a funcionar desde el momento en que se empiezan a recibir datos desde algún cliente o enviar datos hacia uno. Estas aplicaciones son altamente responsivas, resilentes, elásticas y orientadas a mensajes; haciendo que cumplan con el manifiesto publicado por la comunidad en Septiembre del 2014.

Junto al manifiesto reactiva nacieron múltiples framework dispuestos a aplicar lo descrito en este manifiesto, entre ellos Akka, el cual propone un modelo de actores como una manera de abstraer lo necesario e incluso mas, para construir aplicaciones reactivas y con el cual nos basamos para el desarrollo de este proyecto.

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 [1], 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 [2] 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 [3].

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 [4].

## II. Antcedentes

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 [5].

### *II-A. Canales reactivos*

Los canales reactivos son una de las alternativas de comunicación entre dos entidades donde los datos son transmitidos de manera continua sin necesidad de enviar paquetes fijos de datos. Estos son responsivos dependiendo de la cantidad de información que puede ser enviada y recibida.

Al usar los canales reactivos estamos aprovechando la capacidad de hacer back-pressure Fig. 1, 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.

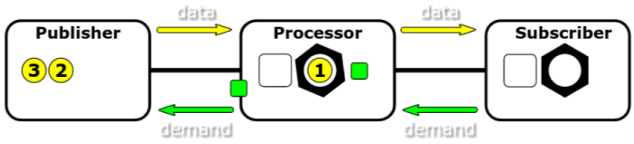


Figura 1. Modelo de comunicación usando Backpressure

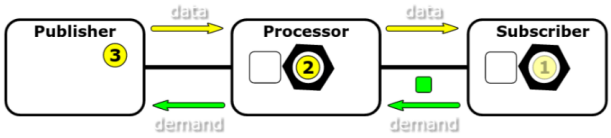


Figura 2.

### *II-B. Modelo de actores*

Es un modelo que permite tener entidades llamadas actores, las cuales facilitaran la programación concurrente y facilitan el envió de mensajes entre otros actores.

De esta manera un actor puede tomar sus propias decisiones, crear mas actores, enviar mensajes, y determinar a quien responder dependiendo del mensaje recibido; son resientes a fallos, lo que significa que se pueden tomar decisiones de acuerdo al fallo presentado, como: reiniciar un actor, crear un nuevo actor, eliminar el actor.

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.4

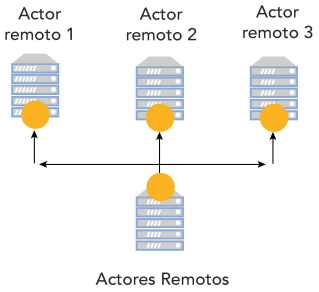


Figura 3. Representación de actores remotos distribuidos

## III. 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.

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

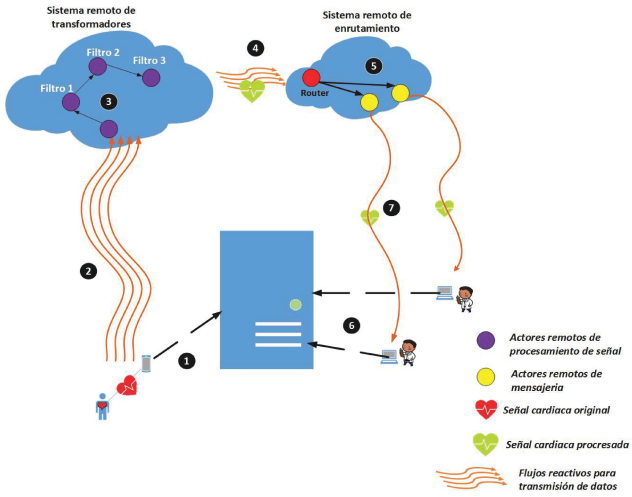


Figura 4. Arquitectura planteada

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 5. Arquitectura planteada

Se crearon dos puntos de acceso, uno era el punto ”Publish”(Fig 5) , 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 5). 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 5.

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 [7], 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 dirección (interfaz)

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

Figura 6. 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 6.

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 7, 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 7. 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. 8. 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.

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

```
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 8. Interfaz implementada por los filtros que manipulan los datos de la señal

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.9.

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

Figura 9. 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.

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. 10.

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 11.

## IV. Pruebas y Experimentos

En la fase de pruebas, se trabajo en la creación de escenarios en los que se pudiera probar la plataforma propuesta VS una

```
$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 10. Código de AngularJs del publisher Web

```
*/
app.controller('chartController', function($scope, $interval) {
    var chart = initGraph();

    var ws = new WebSocket("ws://10.2.67.90:9001/subscribe/1");
    ws.onmessage = function(event) {
        var data = event.data;
        chart.data.datasets[0].data.push(data);
        chart.data.datasets[0].data.shift();
        chart.update();
    }
});
```

Figura 11. Código de AngularJS del cliente observador

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.

**Escenarios**

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**.

Cuadro I. 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 |

**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 12 ).

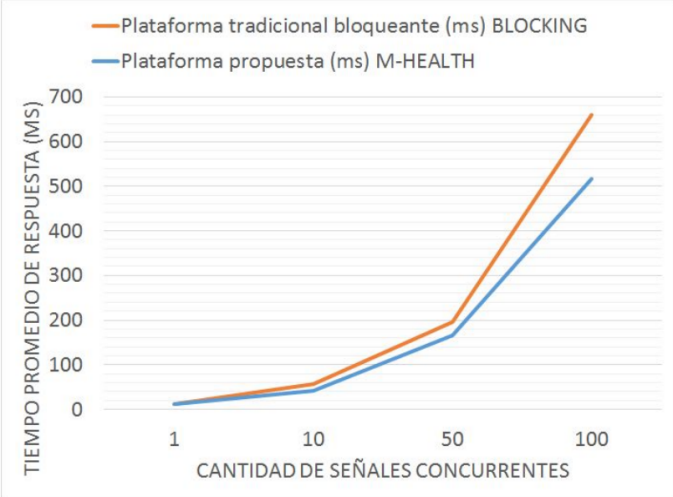


Figura 12. Resultados Escenario 1

**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*

Cuadro II. 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 |

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

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.

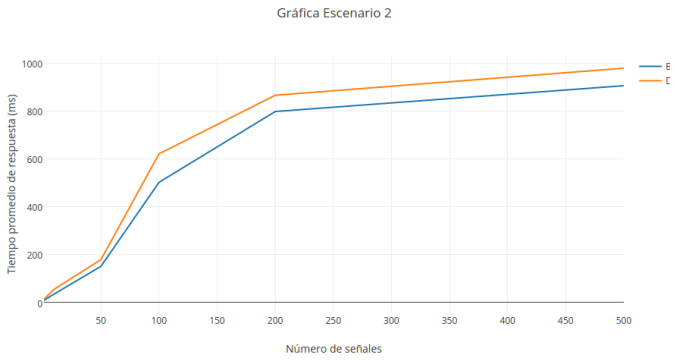


Figura 13. Resultados Escenario 2

## V. CONCLUSIÓN

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.

## RECONOCIMIENTOS

Quisiéramos agradecer con especial énfasis al profesor Hector Fabio Cadavid, quien durante el proceso de investigación estuvo siempre atento a nuestras dudas y consultas; a la decanatura de Ingeniería de sistemas en la Escuela Colombiana de Ingeniería Julio Garavito. Por darnos el espacio y las herramientas necesarias para desarrollar el proyecto.

## REFERENCIAS

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

[2] 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. [Online]. Available: http://publicaciones.unisimonbolivar.edu.co/article/view/536

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

[4] "Remote ECG monitoring system," Patent US3 986 498 A, 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. [Online]. Available: http://www.google.com/patents/US3986498

[5] M. A. C. Acosta and H. F. C. Rengifo, "M-health system backend supported by an actors model," in *Computing Colombian Conference (10CCC), 2015 10th*, pp. 150–156.

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

[7] "Elegant, high-performance http for your akka actors." [Online]. Available: http://spray.io/