ESCUELA COLOMBIANA DE INGENIERA JULIO GARAVITO

Implementación de una arquitectura basada en plataformas de computación de alto desempeño y tiempo real para el procesamiento de señales cardíacas

por

Julián David Devia Serna y Daniela Sepúlveda Alzate

Decanatura

Ingeniera de Sistemas

Directores

Hector Fabio Cadavid Rengifo y Luis Daniel Benavides Navarro

6 de agosto de 2018

ESCUELA COLOMBIANA DE INGENIERA JULIO GARAVITO

Resumen

Ingeniera de Sistemas

por Julián David Devia Serna y Daniela Sepúlveda Alzate

En el proyecto se implementó una arquitectura escalable para el monitoreo en tiempo real de la salud cardíaca (HAAS- Holter As A Service), fuertemente ligada con las tendencias E-Health y M-Health para facilitar el acceso y monitoreo remoto desde dispositivos móviles hacía una plataforma web con acceso de los pacientes, médicos, enfermeras y demás personal médico. Con el objetivo de brindar un mejor servicio a las personas con enfermedades cardíacas, una de las principales causas de mortalidad en el mundo [1].

La arquitectura fue implementada con un framework de software libre llamado Apache Storm e integración de procesamiento con tarjetas de vídeo Nvidia-Cuda, con el objetivo de mejorar el tiempo de ejecución de algoritmos usados para el procesamiento de electrocardiogramas (ECG), que reflejan las señales cardíacas del paciente, también mejorar la capacidad de atención en tiempo real en los centros de salud y la detección automática de cardiopatías mediante la integración de un módulo de Machine Learning.

Agradecimientos

Queremos dar las gracias de manera muy especial, para el decano de ingeniería electrónica Javier Chaparro, por su ayuda en el diseño de los filtros para las señales cardíacas, para Aurora León y Alba Barbosa, por su colaboración en el laboratorio de informática, para nuestras familias, por el amor y la entrega en el desarrollo de nuestras carreras.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Planteamiento del problema

Dos de los mayores retos en la actualidad para Colombia en el sector salud son: el aumento de la cobertura y calidad del sistema de salud y el tratamiento de las enfermedades cardiovasculares, que en el país es una de las principales causas de mortalidad. Las enfermedades cardiovasculares (enfermedades relacionadas con el funcionamiento del corazón y los vasos sanguíneos) fueron cuatro de las diez principales causas de mortalidad en el país para el año 2010, ocupando el primer puesto la enfermedad de isquémica cardiaca, según el Ministerio de salud y protección social (MINSALUD)[2]. Después de siete años las enfermedades cardiovasculares aún están presentes en las principales diez causas de mortalidad de la población colombiana, según un informe preliminar del Departamento Administrativo Nacional de Estadísticas (DANE)[3] para el año 2017 en colombia.

La enfermedad cardiovascular isquémica cardiaca, es la causante del 16,7 % de las muertes de hombres y del 17,2 % de muertes para las mujeres. Esta enfermedad se presenta cuando se ve afectado el flujo sanguíneo al corazón por un bloqueo en las arterias, llegando a ocasionar infartos. En Colombia, se estima según el MINSALUD, que hay en promedio 870 infartos al día y 150 muertes diarias por enfermedades del corazón. Pero esto no es solo un problema que afecta la población colombiana, según un reporte de la OMS (Organización Mundial de la Salud)[4][5], las enfermedades cardiovasculares se encuentran entre el Top 10 de las causas de muerte para el año 2016 a nivel mundial. Entre las dos primeras causas de muerte se encuentra la enfermedad isquémica del corazón y los infartos ocasionando un aproximado de 9,4 y 5,7 millones de muertes respectivamente. Tanto los infartos como la enfermedad isquémica cardiaca, son detectables mediante la lectura de los electrocardiogramas (ECG) por parte del personal médico.

Los ECG son realizados para el monitoreo de la salud cardíaca de los pacientes, mediante un examen llamado Holter, este consiste en la toma de datos cardíacos por periodos de tiempo prolongados, desde doce horas hasta cinco días. En un holter de 48 horas de duración, tomando datos a una frecuencia de 100 datos por segundo, se tienen en total 17.280.000 datos, estos deben ser analizados y procesados para determinar ciertas características que permitan dar un diagnóstico acerca del estado de salud del paciente, su procesamiento puede ser lento y computacionalmente costoso por la cantidad de operaciones que se deben efectuar y los tipos de datos que se deben manejar. Adicionalmente a esto, los hospitales deben realizar el mismo procedimiento para grandes volúmenes de pacientes al mismo tiempo, así que si el mismo examen se realiza con 200 pacientes, en total son más de 3.456 millones de datos que deben ser procesados para poder dar todos los diagnósticos. Normalmente cuando los médicos quieren realizar un examen Holter los pacientes se llevan uno de los monitores de ECG o amanecen en el centro de salud u hospitales por varias horas mientras los datos son recolectados, luego estos son revisados por un médico para dar el resultado y esto puede llevar a un prolongado tiempo de espera para los pacientes.

Este proyecto describe la implementación de una solución escalable, eficiente y reactiva, para solucionar los problemas de identificados, es decir procesamiento de altos volúmenes de datos cardiacos en tiempo real. Con la implementación de la arquitectura, queremos lograr que los pacientes obtengan dispositivos de bajo costo desarrollados por la decanatura de Ingeniería Electrónica como complemento a nuestro proyecto, los cuales transmiten los datos del estado de salud de la persona, ya sea desde la comodidad de su casa o desde el centro de salud. Estos datos son enviados a la arquitectura, la cual se encarga de su visualización en un portal web y su procesamiento en tiempo real para extraer indicadores de la señal o detección de enfermedades cardiacas mediante un módulo de Machine Learning, así si el paciente está presentando alguna alteración se pueda alertar y remitir a la clínica o iniciar algún tratamiento.

1.2. Objetivos del proyecto: General y específicos

1.2.1. Objetivo general

Implementar la prueba de concepto de parte de la arquitectura planteada en el proyecto de Investigación de la Convocatoria Interna de la Escuela "Desarrollo de dispositivos y servicios computacionales para el procesamiento y manipulación automática de datos y señales para el manejo de alteraciones y patologías de los sistemas cardiovasculares y problemas prioritarios de salud".

1.2.2. Objetivos específicos

Implementar la arquitectura planteada haciendo uso de componentes de software libre. Investigar y desarrollar pruebas de concepto de aproximaciones computacionales que permitan el análisis y extracción de características de señales cardíacas en tiempo real.

Identificar e implementar posibles mejoras en complejidad temporal haciendo uso de una infraestructura de computación en paralelo para estas aproximaciones. Integración de un módulo de Machine Learning para la detección de cardiopatías.

1.3. Logros

Durante el desarrollo del proyecto se logró la investigación y formulación del marco teórico, desde el punto de vista fisiológico como técnico para el filtrado, procesamiento y extracción de características de las señales cardiacas a través de la optimización de algoritmos para los Electrocardiogramas (ECG). Luego de la formulación del marco teórico se desarrolló una prueba de concepto, haciendo uso de la plataforma de software libre Apache Storm para computación en paralelo y Hardware Nvidia CUDA para la computación de alto desempeño, con el objetivo de medir la eficiencia del procesamiento del ECG con hasta cinco mil pacientes, los resultados de la prueba de concepto pueden verse en detalle en el capítulo cinco de experimentos.

Se desarrollaron pruebas de escalabilidad de la arquitectura, incluyendo la integración de un módulo de Machine Learning para la detección automática de cardiopatías. Para demostrar esto, se implementó un modelo de Machine Learning para la detección de Apnea. También se desarrolló un visor web para la visualización de la señal cardíaca en tiempo real, de las características e indicadores extraídos de la señal a través de la arquitectura y la clasificación por parte del módulo de Machine Learning. Finalmente, se redactó un artículo en inglés con una breve explicación de la arquitectura implementada.

1.4. Justificación

La popularidad de las tendencias de salud M-Health y E-Health en los últimos años ha llevado al desarrollo de aplicaciones que apoyen a las instituciones para prestar un mejor servicio y brindar más información a sus pacientes acerca de su estado de salud. Con la telemedicina se pueden solucionar los problemas de acceso a los servicios de salud en Colombia y también reducir los índices de mortalidad en pacientes con problemas

cardíacos. El procesamiento de las señales cardíacas de los pacientes exigen de gran capacidad de computo por ser muchos datos, que necesitan ser utilizados para el monitoreo adecuado de la señal cardíaca de una gran cantidad de pacientes, por lo que se requiere una arquitectura que reduzca el tiempo de ejecución de los algoritmos que pueden llegar a ser computacionalmente complejos por la cantidad de operaciones con precisión de punto flotante que se requieren demostrando que una arquitectura escalable configurada con Apache Storm y que aprovecha la computación en paralelo con NVIDIA CUDA permite procesar con un mejor desempeño la información de los electrocardiogramas de múltiples pacientes en tiempo real para la detección de cardiopatías.

1.5. Área de aplicación del resultado del proyecto.

La área de aplicación del producto resultante del proyecto es el sector salud, los hospitles, instituciones prestadoras de salud(IPS), entidades promotoras de salud(EPS), clínicas, centros de salud o cualquier institución u organización interesada en el monitoreo en tiempo real de la salud cardíaca de las personas.

1.6. Organización del trabajo

El libro está organizado de la siguiente manera, un primer capítulo que describe el contexto del problema, la identificación del problema, los objetivos generales y específicos, su justificación, áreas de aplicación del proyecto y justificación. Un segundo capítulo describiendo el estado del arte y el marco teórico del proyecto. Un tercer capítulo, con una visión del producto y su proceso de desarrollo. Un cuarto capítulo, con el detalle de la arquitectura planteada. Un quinto capítulo, que recopila los experimentos de eficiencia y escalabilidad de la arquitectura. Un sexto capítulo, con las conclusiones y trabajos futuros.

Capítulo 2

Marco Teórico y Estado del Arte

2.1. Marco Teórico

El electrocardiograma (ECG) es la visualización de los impulsos eléctricos generados por el flujo de la sangre a través del corazón. Desde el momento en el que entra la sangre al corazón hasta el momento en el que sale se le conoce como ciclo cardíaco. Un ECG se conforma de muchos ciclos cardíacos, los cuales son reflejados como un valor de voltaje correspondiente a un tiempo. En la Fig. 2.1, podemos observar el ciclo de generación de ECG a medida que la sangre pasa por las cavidades del corazón (aurículas y ventrículos).

La forma más común de tomar los electrocardiogramas(ECG), se basa en el método de las doce derivaciones, el cual mide el voltaje del impulso eléctrico que recorre la sangre

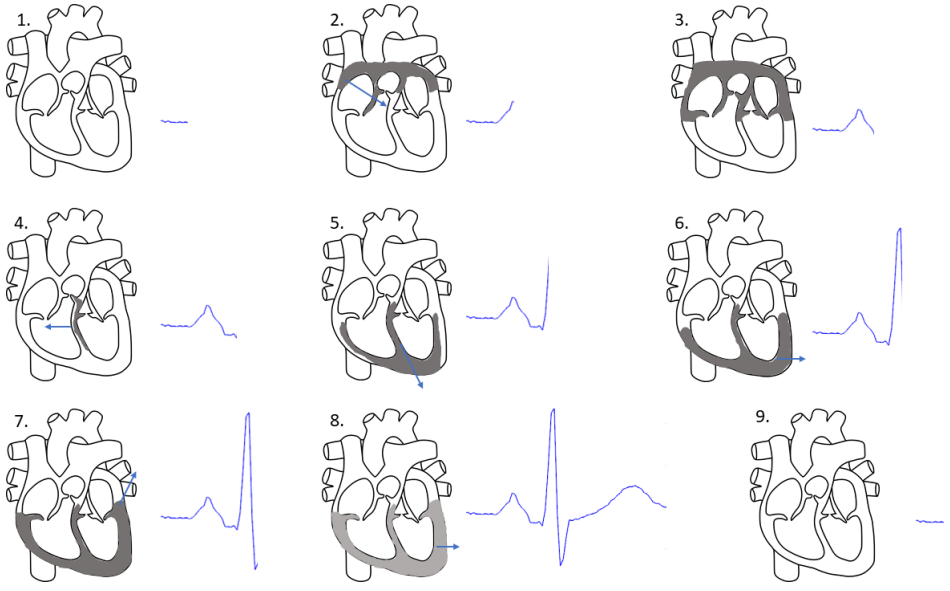


FIGURA 2.1: Fases del ciclo cardiaco

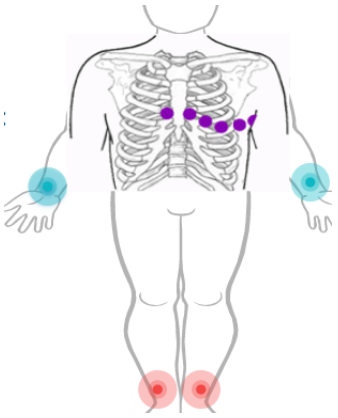


FIGURA 2.2: Ubicación de los electrodos para tomar un electrocardiograma

a través del corazón en doce puntos. Para realizar este análisis es necesario colocar al paciente diez electrodos (Fig. 2.2), cuatro ubicados en sus brazos y piernas, seis colocados al rededor del corazón. Seis de las derivaciones se miden directamente con los electrodos cercanos al corazón, mientras que las otras seis se calculan utilizando los resultados de los otros cuatro electrodos.

En el electrocardiograma (ECG) hay puntos y segmentos claves (Fig. 2.3) que facilitan la formulación de un resultado. En el análisis de un ECG permite a médicos, enfermeros o personal médico reconocer patrones, anomalías o cardiopatías. Algunos de estos puntos y segmentos son:

- Punto P: Despolarización auricular (debajo de 10-15 Hz).
- Complejo QRS: Despolarización ventricular (10-50 Hz).
- Punto T: Repolarización ventricular (debajo de 10 Hz).
- Punto J: Punto en el que el complejo QRS se curva en el segmento ST.
- Segmento ST: Segmento que indica cuando los ventrículos se mantienen en estado despolarizado (puede usarse para identificar condiciones cardiacas).
- Intervalo RR: Longitud del ciclo cardíaco (tiempo entre dos R consecutivas), sirve para medir la variabilidad del ritmo cardíaco.
- Intervalo PQ: Refleja el tiempo de propagación del impulso eléctrico a los ventrículos.
- Intervalo QT: Refleja el tiempo desde la despolarización ventricular hasta la repolarización ventricular.

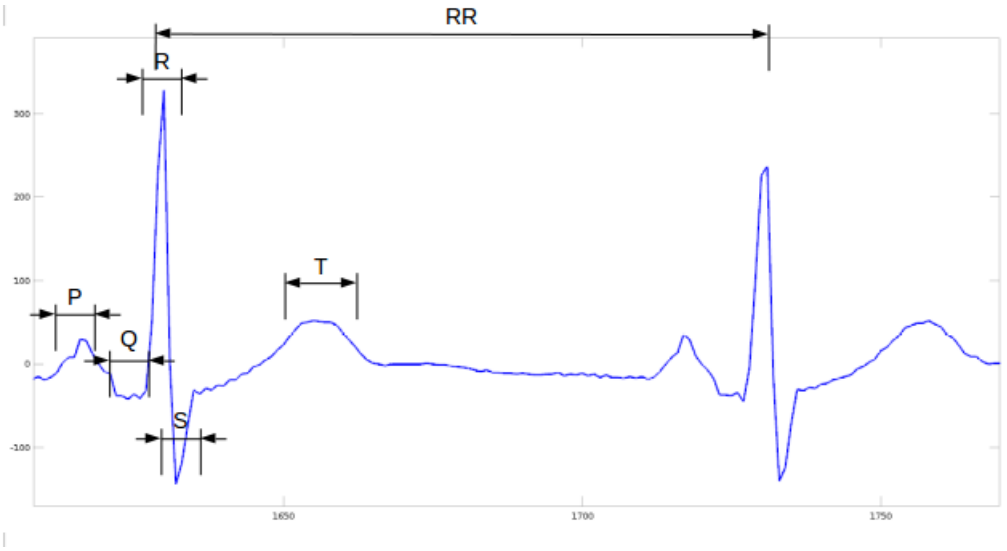


FIGURA 2.3: Puntos clave del ECG

En análisis de un electrocardiograma permite a médicos, enfermeros o personal médico puede reconocer patrones, anomalías o cardiopatías entre ellas la arritmia. La arritmia es la cardiopatía mas común y que se puede identificar fácilmente utilizando el electrocardiograma, la arritmia es una desviación del ritmo cardíaco normal. (50-100 latidos por minuto), esta variación se puede dar de dos formas. Como Bradicardia (ritmo cardíaco mas lento de lo normal) o como Taquicardia (ritmo cardíaco mas rápidos de lo normal), normalmente para identificarlas se miran algunos de los puntos, segmentos mencionados anteriormente. Por ejemplo, para la detección de la arritmia auricular se mira el comportamiento de la onda P, para la detección de la arritmia ventricular se observa el complejo QRS.

Por este motivos es muy importante la nitidez y calidad de la señal cardíaca. Otras cardiopatías que se pueden presentar y se pueden detectar en el electrocardiograma pueden la Isquemia miocárdica(desequilibrio entre la oferta y la demanda de oxígeno del miocardio), se identifica en la inversión de la onda T y el descenso del segmento ST con respecto al punto J. El Infarto miocárdico(Muerte de una parte del miocardio por falta de sangre) se identifica por una onda R reducida y una onda Q más larga.

Cuando se toman los datos de un electrocardiograma (ECG) se utilizan unos electrodos sobre la superficie de la piel. Estos electrodos, puede presentar anomalías causadas por factores externos llamadas ruido o artefactos, estas anomalías pueden darse por diferentes causas como la línea base extraviada, es una baja frecuencia extraña que puede resultar por respiración, movimiento, mal contacto de los electrodos (frecuencia menor a 1Hz), el artefacto causado por el de movimiento de electrodos (frecuencia entre 1-10 Hz), la interferencia de línea de energía (frecuencia mayor a 50-60 Hz) puede ser causada por interferencia con otros equipos o por una mala conexión a tierra, ruido electromiográfico,

que se presenta durante el ejercicio por movimientos bruscos o la actividad Respiratoria, causa ruido debido a los cambios en el pecho, cambios en la posición del corazón y la conductividad de los pulmones.

Cuando suceden algunas de las anomalías mencionadas anteriormente es necesario filtrar y limpiar la señal antes de ser procesada para evitar incluir en ella datos que puedan alterar el resultado o el análisis del paciente y poder realizar un análisis computacional del electrocardiograma. Luego de ser filtrado removiendo ruidos y artefactos, es necesario identificar el complejo QRS, con el que se pueden identificar diferentes anomalías y hallar el ritmo cardiaco y variabilidad (principales indicadores del estado del corazón), entre otros indicadores y características.

2.2. Estado del arte

Se han desarrollado anteriormente trabajos que exploran el uso de Apache Storm para procesar grandes cantidades de datos recibidos por monitores cardíacos, esto gracias a la escalabilidad y fácil configuración que ofrece. En estos casos se configura en la nube para recibir los datos enviados por sensores y dispositivos móviles. En trabajos como [6] y [7] se muestra la aplicación de Apache Storm para este tipo de aplicaciones, pero sin contemplar el uso de plataformas de alto desempeño para mejorar el tiempo de procesamiento de los ECG.

Otros trabajos se han enfocado en proponer arquitecturas con sensores y una infraestructura en la nube, para el procesamiento y análisis de datos de salud recolectados por los sensores, como se muestra en [8] y [9], donde las propuestas hacen énfasis en la importancia de la conexión entre los sensores, celulares, wearables y la infraestructura encargada de realizar el trabajo de procesamiento de los datos y nosotros complementamos esto, enfocándonos en el procesamiento de los datos con una arquitectura de software para procesamiento altamente eficiente y con la posibilidad de conectar un módulo de machine learning o deep learning para realizar clasificación de los datos del paciente.

También se ha realizado mucha investigación acerca de los sensores que se usan para recolectar datos fisiológicos con la ayuda de herramientas de monitoreo y la creciente tendencia de IoT. En [10] se muestra un avance acerca de un protocolo diseñado para transmitir los datos de un ECG con la menor latencia y el menor impacto en la calidad de la señal para ayudar a los servicios de emergencia a tener la información en tiempo real.

Por otro lado en [11] la investigación se orienta hacia el diseño de sensores que se encarguen de procesar una parte de la señal y de esta forma que el transporte sea mas

sencillo y liviano para ahorrar energía y tiempo. También se encuentra el trabajo de [12] en la creación de wearables que transmitan la información por wifi para el monitoreo del paciente. También se complementa eso con la investigación acerca de wearables 2.0 en [13] donde se considera el hecho de que los sensores deberían ser cómodos para los usuarios y diferentes de acuerdo a cada uno, también se habla acerca de que el diseño debería ir orientado hacia smart clothing es decir prendas de vestir como camisetas que pueda estar interconectada con la red para poder comunicar datos capturados acerca de la salud y los hábitos de los usuarios.

Con el creciente interés de las personas en su salud y el uso de dispositivos móviles como celulares, tablets, smartwatches, etc, se han generado trabajos como el que se muestra en [14], donde se desarrolla una aplicación móvil para visualizar el ECG y el ritmo cardíaco capturados por un sensor y transmitidos vía Bluetooth, para luego enviar datos estructurados a la nube donde serán procesados.

En [15] se habla acerca de un sistema para recolectar y procesar los datos de un conjunto de sensores y disparar alertas cuando estos valores superan algunos límites establecidos para informar al personal encargado acerca del paciente. También hay estudios acerca de la detección de anomalías de formas mas elaboradas como el análisis de la morfología del corazón para buscar indicadores y características que indiquen enfermedades o anomalías, métodos como detección de picos o amplitudes en las ondas del ECG como se muestra en [16].

Capítulo 3

Visión del producto

3.1. Visión del producto

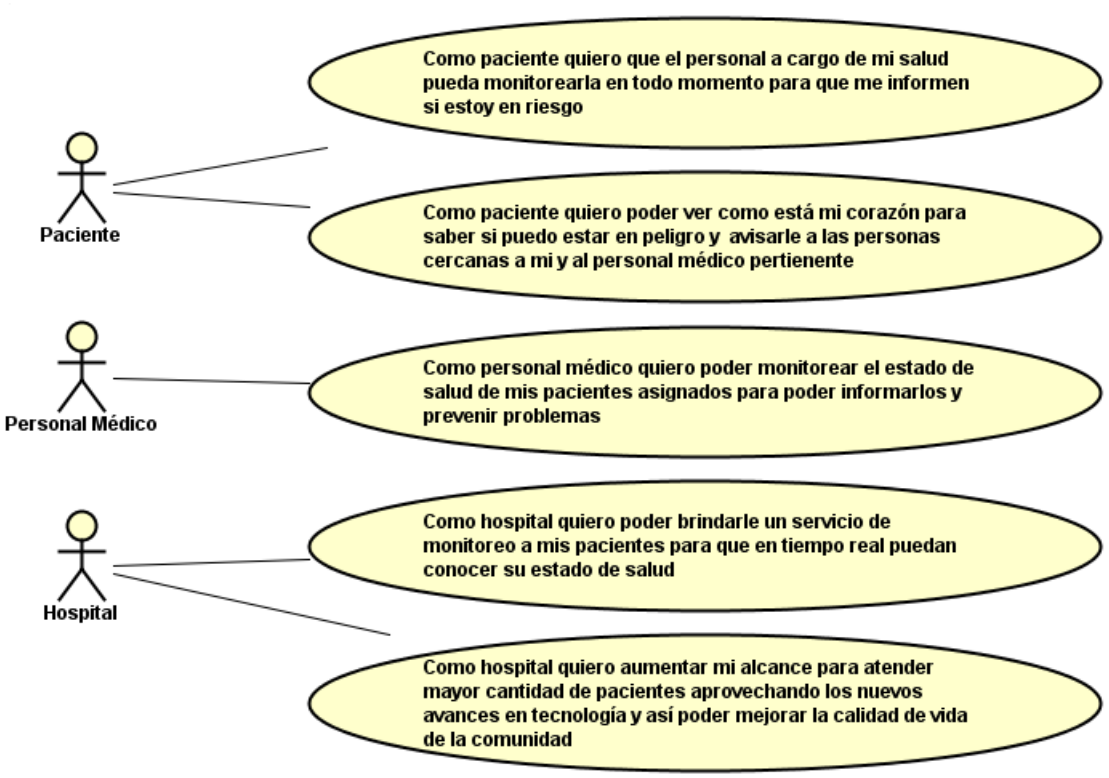


FIGURA 3.1: Historias de usuario de la arquitectura

Como se ve en la Fig. 3.1, para este caso se tienen 3 actores que pueden sacar provecho de esta arquitectura, se encuentran los pacientes interesados en cuidar su salud, y que usarían la arquitectura para poder ver ellos mismos y el personal médico pertinente, también se encuentra el personal médico que quiere poder saber como se encuentran sus pacientes y poder alertarlos en caso de que se presente una irregularidad y por

último están los hospitales, IPS, EPS, clínicas, y cualquier institución interesada en cuidar y atender la salud de un grupo de personas, estas necesitan aumentar su alcance permitiendo que mas personas usen sus servicios para prevenir accidentes y muertes, así como también necesitan el monitoreo constante de los pacientes para saber cuando algo está ocurriendo con ellos, estos son los principales problemas y necesidades que está plataforma busca resolver al permitir el monitoreo en tiempo real de grandes volúmenes de pacientes.

3.2. Proceso de desarrollo

Se utilizó la metodología SCRUM con sprints de 3 semanas, para realizar las tareas necesarias para el desarrollo del proyecto, en el transcurso de 7 meses comenzando en enero y terminando en Agosto. Al finalizar cada sprint se definía que tareas era necesario pulir y refinar junto con el desarrollo de las actividades del nuevo sprint. Las actividades realizadas en cada sprint se detallarán mas adelante en el Apéndice A.

Inicialmente se buscó conseguir la información correspondiente al marco teórico en el que primero se trabajó en entender la fisiología del corazón y su funcionamiento, luego se indagó en el funcionamiento de los electrocardiogramas para seguir con los conceptos básicos de procesamiento de señales como el filtrado y los algoritmos para extracción de características concretas de los electrocardiogramas. Luego se buscó el estado del arte acerca del proyecto, buscando información acerca de plataformas para el procesamiento de señales y el trabajo que se ha hecho usando Apache Storm para procesar electrocardiogramas, así mismo con plataformas de alto desempeño y en concreto Nvidia Cuda. Teniendo el estado del arte y el marco teórico se realizó la planeación de los componentes que se tendrían en cuenta y se comenzó la familiarización con las herramientas y los modelos de programación de cada una. El siguiente aspecto a abordar fue la implementación de los componentes con los algoritmos estudiados y las pruebas de rendimiento para medir el desempeño de la arquitectura, con los componentes implementados para la extracción de características.

Con la arquitectura completa se realizó la investigación de bases de datos especializadas para tomar muestras y sobre modelos de machine learning que pudieran utilizarse para clasificar este tipo de datos, cuando se determinó la base de datos a utilizar y el modelo de machine learning se implementó utilizando Weka y se conectó con la arquitectura para realizar la clasificación. Finalmente se desarrolló un visor web para la visualización de los resultados entregados luego del procesamiento realizado por la arquitectura y se integró con esta por medio del protocolo STOMP.

Capítulo 4

Arquitectura de la solución

4.1. Arquitectura propuesta

Proponemos una arquitectura altamente escalable que integra plataformas de computación en tiempo real y de alto desempeño (usadas para BigData y computación en GPU respectivamente) para minimizar el tiempo de procesamiento de grandes cantidades de flujos de señales de salud. La arquitectura está compuesta por unidades procesamiento como filters, functions y monitors (Fig. 4.1). Los filters transforman la señal , las functions extraen características de la señal que reciben y los monitors almacenan o transfieren información a componentes de entrada o salida (componentes de software, visores, discos duros, etc.). Estas unidades de procesamiento usan diferentes métodos de composición como pipe, branch y parallel (Fig. 4.1). Pipe se encarga de encadenar dos unidades de procesamiento de forma que la salida de la primera, es la entrada de la segunda, branch se encarga de repartir la salida de una unidad de procesamiento para que la reciban dos o más unidades de procesamiento y parallel transforma filtros y funciones normales en algoritmos que hagan uso de computación de alto desempeño (HPC) para mejorar su tiempo de ejecución, esto se realiza concretamente en infraestructuras que cuenten con GPU.

Un ejemplo de una arquitectura diseñada usando estos métodos de composición y unidades de procesamiento se muestra en la Fig. 4.2. Esta figura muestra a la izquierda múltiples dispositivos que envían señales (dispositivos como bandas, celulares o sensores de otra índole), esta señal es recibida por los endpoints, los cuales comienzan un flujo de procesamiento por señal.

En el ejemplo de la arquitectura en la Fig. 4.2, la señal es filtrada usando el filter1 y esta se divide por medio de un branch hacia function1 y una parallel function2. La

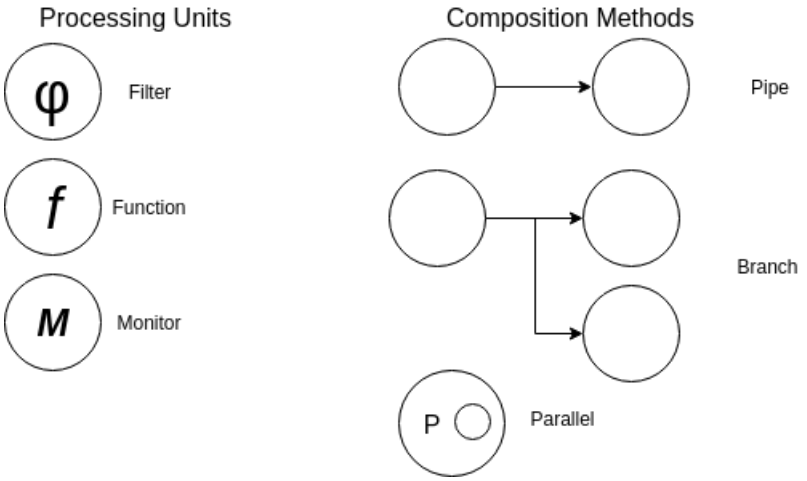


FIGURA 4.1: Unidades de procesamiento y métodos de composición de la arquitectura

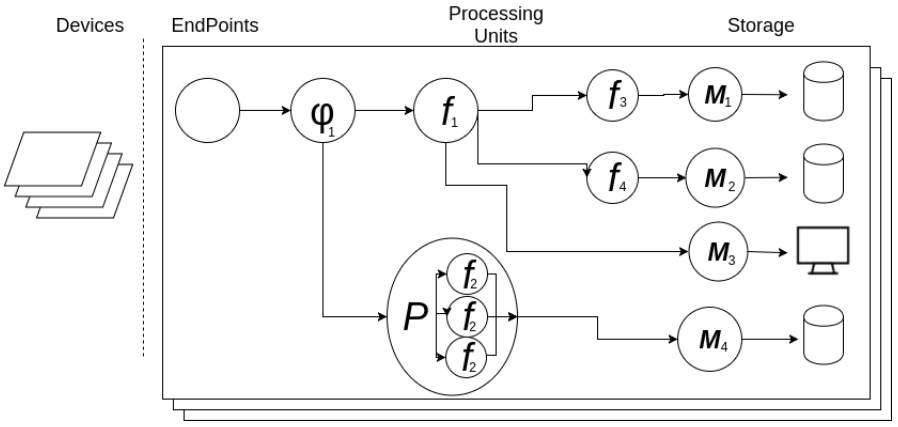


FIGURA 4.2: Ejemplo de arquitectura

function2 se encadena con el monitor4, el cual almacena el resultado. La function1 envía su resultado a function3 y function4 y al monitor3 para mostrar su visualización mientras que function3 y function4 envían sus resultados a monitor1 y monitor2 respectivamente para almacenar los resultados.

Utilizamos Apache Storm y Nvidia CUDA para implementar un ejemplo concreto de los componentes descritos en la arquitectura. Apache Storm es un framework de software libre para el procesamiento de flujos de datos en tiempo real, trabaja con una estructura de datos llamada topología, la cual representa un grafo acíclico dirigido (DAG), los nodos de este grafo son clasificados como spouts o bolts. Los spouts son los nodos responsables de recibir los datos de fuentes externas como Redis, MQTT, Kafka, etc. Estos spouts son lo que en la arquitectura se mostró como endpoints. Los bolts son las unidades de procesamiento a cargo de desarrollar una tarea con los datos que reciben, estos se encargaran de lo que en la arquitectura se describió como filters, functions y monitors. Los spouts y bolts se comunican por medio de streams, los cuales no son más que una

secuencia de tuplas que llevan los datos necesarios. En este caso cada tupla del stream lleva una porción del electrocardiograma (ECG).

Para la configuración de Apache Storm en modo cluster, se manejan tres servicios. El servicio Nimbus (que cumple el rol de nodo maestro del cluster) es responsable de asignar tareas, distribuir datos y monitorear posibles fallas en los supervisores. El servicio de Zookeeper (en un nodo trabajador del cluster) está a cargo de coordinar los nodos del cluster, monitorear a los supervisores, compartir y mantener la información sincronizada. Y el servicio de Supervisor (en un nodo trabajador) que es responsable de desarrollar las tareas asignadas por el Nimbus de acuerdo a la configuración de la topología. Por otro lado, utilizamos el API de Nvidia Cuda el cual nos permite utilizar los núcleos de las GPU para ejecutar algoritmos en paralelo. Para este proyecto se utilizó un cluster con 4 equipos, un zookeeper, un nimbus y 2 supervisores.

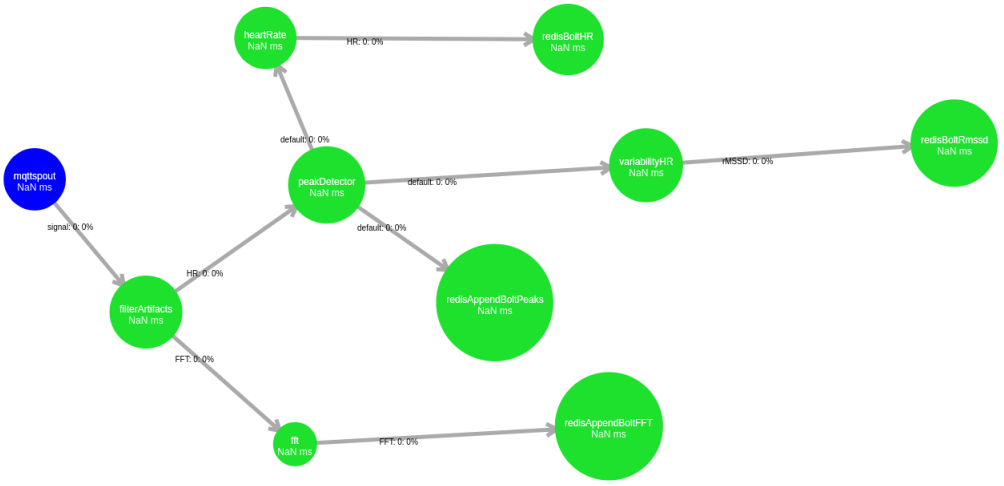


FIGURA 4.3: Implementación concreta de la topología mostrada en Fig. 4.2 con Apache Storm

la Fig. 4.3 muestra una implementación en storm de la arquitectura planteada en Fig. 4.2 y sus componentes se describen a continuación: El spout recibe los datos a través de MQTT, el cual es un protocolo de mensajería liviano que funciona bajo el patrón de publicar/suscribir, es ampliamente utilizado en el campo de aplicaciones relacionadas con IoT. El spout envía los datos del ECG al bolt filterArtifacts, el cual se encarga de realizar el filtro de suavizado y el de 17 Hz, el resultado del filtro de suavizado lo recibe el bolt fft que se encarga de calcular la frecuencia significativa resultado de la FFT, mientras que el resultado del filtro de 17 Hz es recibido por el bolt peakDetector, el cual ubica los picos R del ECG y envía esta información a los bolts heartRate y variabilityHR para que calculen el ritmo cardíaco y los indicadores de variabilidad correspondientemente.

4.1.1. Algoritmos utilizados

4.1.1.1. Filtro de Suavizado

El filtro de suavizado se encarga de remover la interferencia, artefactos y de ubicar todo el ECG sobre la misma lìnea base. En este caso se usó un filtro pasa altos de 0.05Hz de orden 19 para la señal del ECG tomada con una frecuencia de muestreo de 100Hz. El filtro fue implementado con la siguiente sumatoria, donde Y representa la señal filtrada, X el ECG sin filtrar y C es la constante de filtrado:

$$Y(i) = \sum_{k=1}^{19} X(i-k) * C(k)$$

En la Fig. 4.3 este filtro está implementado en el bolt llamado FilterArtifacts.

4.1.1.2. Filtro de 17Hz

Este filtro limpia la señal de ruidos y artefactos que puedan distorsionar las ondas QRS. También reduce las ondas P y T para resaltar el complejo QRS, el cual se usa para calcular el ritmo cardíaco. En este caso se usó un filtro pasa banda con un límite inferior de 13Hz y un límite superior de 21Hz de orden 19 para la señal del ECG tomada con una frecuencia de muestreo de 100Hz. El filtro fue implementado con la siguiente sumatoria, donde Y representa la señal filtrada, X el ECG sin filtrar y C es la constante de filtrado:

$$Y(i) = \sum_{k=1}^{19} X(i-k) * C(k)$$

Como se puede ver en las sumatorias anteriores (filtro de suavizado y de 17Hz), para calcular cada valor, es necesario conocer los anteriores 19 valores, esto significa que los primeros 19 elementos no pueden ser filtrados. Este filtro al tener la misma implementación que el anterior pero con diferentes constantes de filtrado, se efectuó también en el bolt llamado FilterArtifacts en Fig. 4.3.

4.1.1.3. Detector de picos

Este procedimiento tiene el objetivo de detectar los puntos más altos de cada complejo QRS (el punto R). Detectar este punto permite calcular el ritmo cardíaco y su variabilidad.

En este caso se calcula primero una media dinámica cada 100 valores (por la frecuencia de muestreo del ECG), de estos valores se encuentra el más alto, este se multiplica por una constante dada para esta frecuencia de ECG (en este caso la constante es 10, descubierta empíricamente) para establecer un límite y eliminar los valores inferiores a este. Los valores restantes serán la parte superior del complejo QRS, luego de esto, es necesario seleccionar solo el punto más alto de cada complejo QRS, esto se realiza utilizando el concepto de derivada. Este algoritmo se implementó en el bolt llamado peakDetector en la Fig. 4.3.

4.1.1.4. Ritmo Cardíaco

Para calcular el ritmo cardíaco se necesita conocer la ubicación de todos los picos R, una vez estos son identificados como se expone en el punto 3.2.1.3. El ritmo cardíaco se calcula con la diferencia de las ubicaciones sucesivas de estos picos, la cual después se multiplica por 60 y se divide entre la frecuencia de la señal (100 en este caso) para dejar la unidad de medida en latidos por minuto. Esto permite realizar proyecciones de lo que sería el ritmo cardíaco si se mantuviera ese intervalo de tiempo entre cada pico R, y de estos valores se tienen en cuenta el promedio, el mínimo y el máximo. En la Fig. 4.3 esto se encuentra en el bolt llamado heartRate.

4.1.1.5. Variabilidad

Este algoritmo busca identificar algunos indicadores que se mostrarán a continuación acerca de la forma como varía el ritmo cardíaco utilizando las ubicaciones de los picos R detectados anteriormente. Fue implementado en el bolt variabilityHR en la Fig. 4.3.

- SDNN: Desviación estandar de los intervalos R-R.
- rMSSD: Raíz cuadrada de las diferencias sucesivas de intervalos N-N.
- NN50: Parejas de intervalos R-R separados por mas de 50ms.
- NN20: Parejas de intervalos R-R separados por mas de 20ms.
- pNN50: Porcentaje de NN20.
- pNN20: Porcentaje de NN50.

4.1.1.6. Transformada rápida de Fourier

La transformada rápida de fourier (FFT) es usada para extraer la frecuencia con mayor cantidad de ocurrencias, entre las presentes en el ECG. el algoritmo se encarga de descomponer la señal en diferentes funciones, cada una representando una frecuencia, esta información se puede utilizar para detectar si hay algo que no se encuentra normal al encontrarse frecuencias poco comunes en el ECG. Este algoritmo por la complejidad que tiene se desarrolló utilizando la biblioteca cuFFT de Nvidia CUDA para mejorar el tiempo de ejecución al efectuar su cálculo de forma paralela. Se utilizó la transformada que recibe números reales y retorna números complejos, a estos números complejos se les calculó el cuadrado de su magnitud y donde se encontrara el mayor resultado, esta sería la frecuencia con mayor número de ocurrencias. Esta implementación se realizó en el bolt llamado fft en la Fig. 4.3.

4.1.2. Escalabilidad

En la interfaz web de Apache Storm se puede ver la distribución que éste le da a los bolts y spouts (Fig. 4.4.) de acuerdo a la configuración que se le dió a la topología en los supervisores disponibles permitiendo con solo cambiar la configuración de la topología aumentar la cantidad de instancias de determinado bolt o spout

FIGURA 4.4: Muestra de la distribución de los componentes de la topología en Apache Storm

4.2. Modelos de datos

4.2.1. Modelo REDIS

Redis es una estructura de datos en memoria que se utiliza como caché, base de datos o broker de mensajería, en este proyecto se utilizó como caché y base de datos, bajo un esquema de llave valor. en la tabla 4.1 se presentarán las llaves, el tipo de datos y la función que tenía cada llave que se utilizó en este proyecto para la función de caché y en la tabla 4.2 las que se usaron con la función de base de datos, debido a que en el proyecto se utilizaban llaves distintas para cada paciente diferenciándose con el id del paciente, este id se reemplazará con el carácter "$".

Llave	Tipo de datos	Función
$c	List	Caché para filtro de suavizado y 17Hz
$DataCollector	List	Caché para recolector de características
$multiFFT	List	Caché para acumulador de datos para transformada múltiple de Fourier

Cuadro 4.1: Tabla con las llaves usadas de Redis para la función de caché

Llave	Tipo de datos	Función
$artifacts	List	ECG suavizado
$hrfilter	List	ECG filtro de 17 Hz
$frequency	List	Frecuencia significativa del ECG
$FFTR	List	Resultado FFT
$peaks	List	Ubicación de los picos en el ECG
$hr	List	Ritmo cardíaco promedio
$hrmin	List	Ritmo cardíaco mínimo
$hrmax	List	Ritmo cardíaco máximo
$sdnn	List	SDNN
$rmssd	List	rMSSD
$nn50	List	nn50
$nn20	List	nn20
$pnn50	List	pnn50
$pnn20	List	pnn20

Cuadro 4.2: Tabla con las llaves usadas de Redis para la función de base de datos

4.3. Recursos (Tópicos MQTT/Tópicos STOMP)

Se utiliza el protocolo MQTT para recibir los datos de los pacientes para ser procesados y el protocolo STOMP para enviar la información a su visualización, los tópicos están

distribuidos para cada paciente. A continuación se listaran esos tópicos y los tipos de datos que estos manejan, se reemplazará el id del paciente con el carácter "$"

4.3.1. Tópicos MQTT

Se utiliza el protocolo MQTT por el bajo peso de los mensajes y la facilidad que esto implica para su uso en aplicaciones de IoT, este funciona bajo el patrón de publish/subscribe bajo un esquema de tópicos, los tópicos de MQTT que se utilizaron se encuentran en la tabla 4.3.

Tópico	Función	Tipo de dato que recibe
paciente/$	Recepción de datos paciente	Los N voltajes del ECG separados por espacio (actualmente se tiene configurado todo para que funcione con N=512)

Cuadro 4.3: Tabla con los tópicos de MQTT que se utilizaron

4.3.2. Tópicos STOMP

Se utiliza el protocolo STOMP por su sencillez y su alto uso en aplicaciones web, este funciona bajo el patrón de publish/subscribe bajo un esquema de tópicos, los tópicos de STOMP que se utilizaron se encuentran en la tabla 4.4.

Tópico	Función	Tipo de dato que envía
/topic/$_viewer	Datos del ECG suavizado para visualización	la frecuencia de muestreo, junto con cada dato el ECG enviado individualmente separados por : de la forma frec:dato
/topic/$_data	Características del ECG	Una cadena que representa un arreglo con todas las características extraídas en forma de llave valor (llave:valor) separadas por coma e incluyendo un [al comienzo y un] al final. Las características son time (tiempo de llegada de los datos), HR (ritmo cardíaco promedio), HRMAX (ritmo cardíaco máximo), HRMIN (ritmo cardíaco mínimo), pNN50, pNN20, NN20, NN50, SDNN, rMSSD, FFT (frecuencia significativa del ECG extraída de la FFT), CLASS (clasificación dada por el modelo de Machine Learning, A si es Apnea y N si es Normal).

Cuadro 4.4: Tabla con los tópicos de STOMP que se utilizaron

4.4. Vistas arquitectónicas

4.4.1. Vista lógica

4.4.1.1. Modelo de componentes

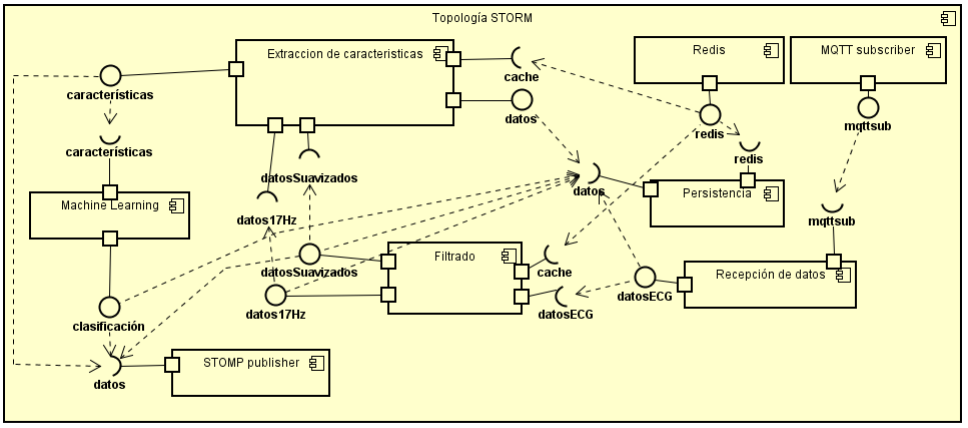


FIGURA 4.5: Diagrama de componentes de la topología de Storm

En el diagrama de componentes mostrado en la Fig. 4.5 se detallan los componentes que conforman la topología de Storm para procesar el ECG, se necesita inicialmente un componente que permita realizar suscripciones a un tópico de MQTT, este se conecta con un componente de Recepción de datos que una vez recibe el ECG, crea el flujo de procesamiento para ese paciente. Con fines de persistencia y caché, se necesita un componente que permita conectarse con Redis para guardar y consultar información. Una vez se tiene la información el componente de filtrado haciendo uso del caché, se encarga de remover los ruidos y artefactos presentes en el ECG, así como también pasar un filtro de 17Hz, estos resultados, tanto el ECG suavizado como el filtrado a 17Hz son enviados al extractor de características que hace uso del componente de Redis para caché y del componente de persistencia, luego de extraer las características necesarias, estas son enviadas al componente de Machine Learning para que realice su clasificación y envío al componente encargado de publicarlo en STOMP para que sea recibido por el visor web.

El componente de extracción de características mostrado en la Fig. 4.6 requiere de otros componentes que le provean acceso al caché y el ECG suavizado y filtrado a 17Hz, el suavizado se pasa por el componente de transformada de Fourier que hace uso del API de NVIDIA Cuda para realizar la FFT en paralelo, luego el componente detector de frecuencia significativa se hace cargo de encontrar cual es la frecuencia mas relevante del ECG y esta información la recibe el componente recolector de características quien hace

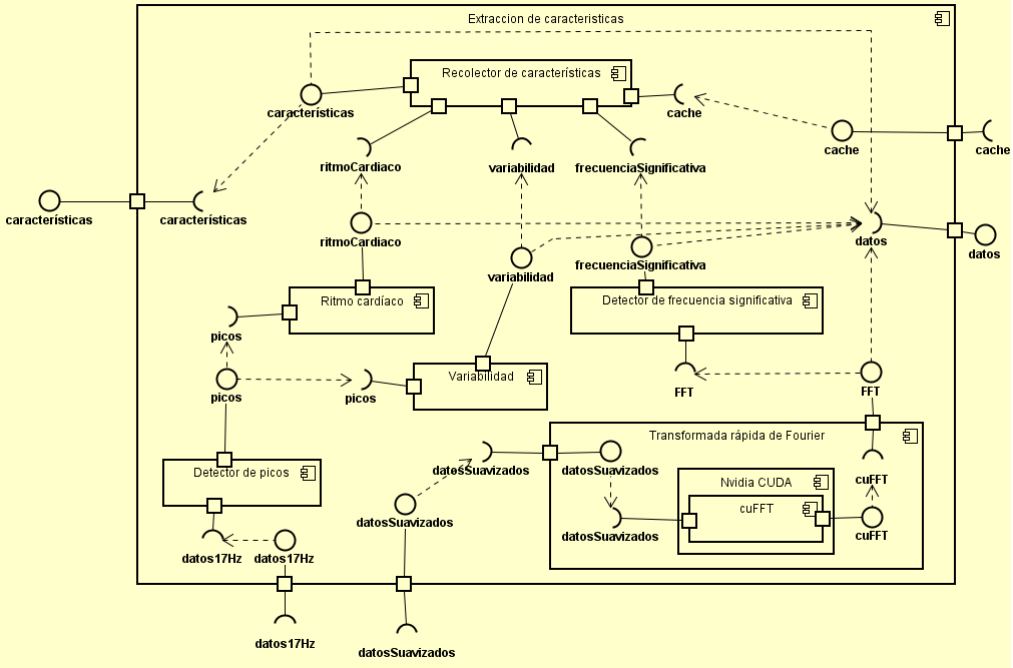


FIGURA 4.6: Detalle del componente extractpr de características de la topología

uso del caché. Al mismo tiempo el ECG filtrado a 17 Hz es utilizado por el componente de detección de picos que le entrega la ubicación de los picos en el ECG a los componentes de ritmo cardíaco y variabilidad, los cuales entregan sus resultados también al recolector de características, una vez se tienen todas las características completas, estas son las que recibe el componente de Machine Learning.

4.4.1.2. Modelo de integración Storm-CUDA

4.4.2. Vista física

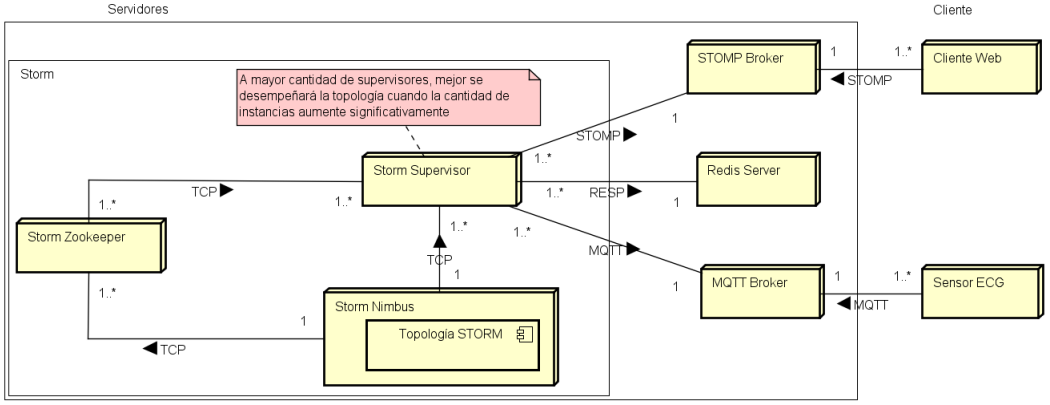


FIGURA 4.7: Diagrama de despliegue de la arquitectura

Inicialmente es necesario que el paciente cuente con un dispositivo que se encargue de capturar el ECG, este dispositivo por medio del protocolo MQTT envía la información al servidor que actúa como Broker MQTT para que este la envíe al Supervisor de Storm encargado de tener la función de spout (para esto es necesario contar con un Storm Nimbus y un Zookeeper que se encarguen de mantener la comunicación entre todos los supervisores), en el paso del ECG por la topología se utilizará el servidor Redis para el caché y para el almacenamiento. Posteriormente cuando todas las características del ECG sean extraídas y se haya clasificado entre Apnea y Normal, esta información se enviará al Broker STOMP para que por medio del protocolo STOMP haga llegar esta información al Cliente Web para que el paciente y los doctores involucrados puedan tener acceso. Esto se ve en el diagrama de despliegue de la Fig. 4.7 que muestra los componentes físicos de la arquitectura y la forma en la que estos interactúan.

4.4.3. Vista dinámica

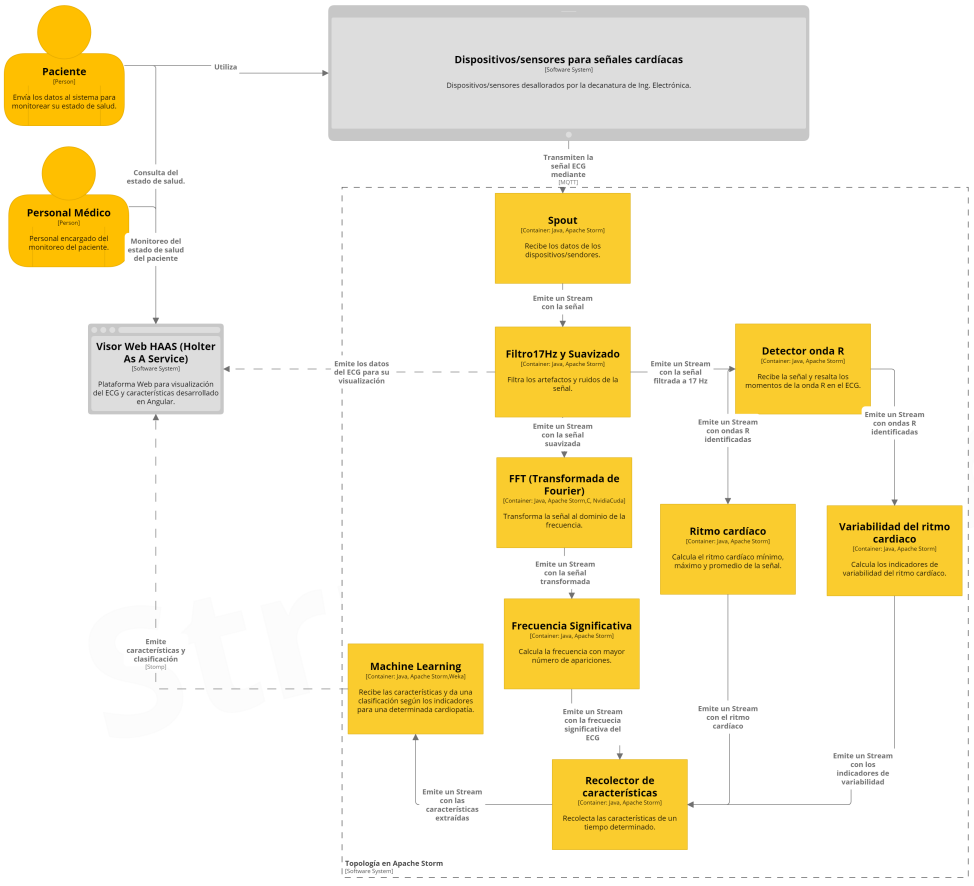


FIGURA 4.8: Diagrama que muestra la interacción entre los actores y el la arquitectura implementada

En esta arquitectura hay dos tipos de usuarios, los pacientes y el personal médico, los pacientes utilizan los dispositivos que capturan los datos del ECG y los envían por

medio de MQTT los hacen llegar al spout, este los transmite a la fase de filtrado que a su vez los envía al visor web (el ECG suavizado), a la transformada de fourier (el ECG suavizado) y a la detección de picos (el ECG filtrado a 17Hz), el detector de picos envía las ubicaciones de estos picos al detector de ritmo cardíaco y al calculo de variabilidad del ritmo cardíaco, estos una vez tienen las características necesarias, las transfieren al recolector de características. Mientras esto sucede la transformada de fourier envía su resultado al detector de frecuencia significativa y una vez este ha completado su tarea, envía el resultado al recolector de características, que cuando completa todas las características que espera, las envía al machine learning para que este clasifique el ECG en Apnea o Normal y envíe estas características con su clasificación al visor web, donde tanto el paciente como el personal médico pueden monitorear los resultados. Este comportamiento se encuentra en la Fig. 4.8[1].

[1]Enlace del diagrama en structurizr:
https://structurizr.com/share/39801/diagrams#Container%20Apache%20Storm

Capítulo 5

Experimentos

5.0.1. Escalabilidad de Apache Storm

Para validar la eficiencia y desempeño de la arquitectura planteada, se desarrollaron dos pruebas. Con el fin de conocer el efecto que tiene escalar la cantidad de instancias de spouts y bolts en la topología, se midió el tiempo que demora en tener el resultado de procesar los datos, desde el momento en que el spout los recibe, hasta que los bolts los envían a redis. En la Fig. 5.1 el eje X muestra la cantidad de spouts-bolts con los que se hicieron, entonces 1-1, significa 1 instancia del spout y 1 de cada bolt, 1-5, es 1 instancia del spout y 5 de cada bolt y así sucesivamente. Mientras que en el eje Y se encuentra el tiempo en milisegundos que se demoró en dar el resultado, donde el color azul representa el cálculo del ritmo cardíaco, y el rojo los indicadores de variabilidad del ritmo cardíaco.

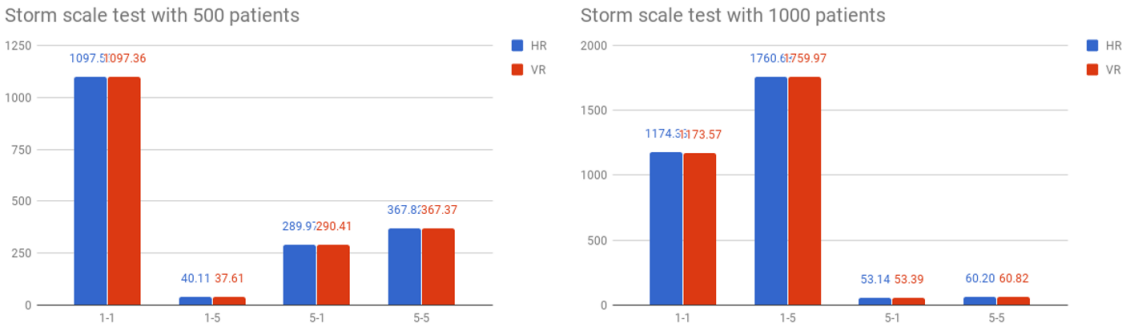


FIGURA 5.1: Medida del tiempo de procesamiento para 500 y 1000 pacientes

En los gráficos de la Fig. 5.1 se pueden observar que el hecho de aumentar la cantidad de instancias de los bolts refleja un mejor desempeño para 500 pacientes, mientras que cuando esta cifra aumenta a 1000, lo que resulta clave es aumentar la cantidad de instancias del spout que recibe los datos.

5.0.2. Prueba de eficiencia con Nvidia CUDA

El objetivo de esta prueba es comparar el desempeño del algoritmo FFT de manera lineal, con la ejecución en paralelo de Nvidia CUDA, teniendo en cuenta que el algoritmo de CUDA se ejecuta en la versión de C del api CUDA, en concreto se utilizó cuFFT, para hacer una comparación justa, en la versión lineal se utilizó la biblioteca FFTW de C++ que es la mas popular para este propósito.

En esta prueba se hicieron dos versiones, la primera fue ejecutando una transformada de Fourier para conjuntos de datos con cantidades variando desde 256 hasta 131072 y la segunda versión fue realizando varias transformadas de fourier juntas, se realizaron desde 2 hasta 256 transformadas y los resultados se muestran en la Fig. 5.2, en donde se ve que en la primera versión del experimento el tiempo en milisegundos de CUDA siempre fue mayor al tiempo de C++, mostrando que en este caso el paralelismo no reflejaba una mejora considerable en este caso, pero en la segunda versión del experimento, el rendimiento de C al ejecutar 128 transformadas de Fourier para 512 datos es bastante similar al de CUDA y cuando se llega a 256 transformadas, el programa en C no las puede procesar mientras que el de CUDA si, en este caso se ve que el paralelismo masivo de CUDA es donde se encuentra la ventaja de este modelo de programación

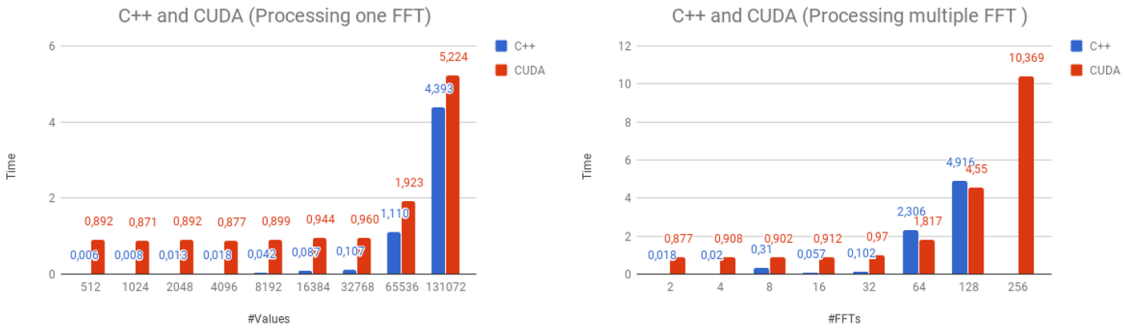


FIGURA 5.2: Medida de tiempo ejecutando una FFT y múltiples FFTs

Capítulo 6

Conclusiones

6.1. Conclusiones

En las pruebas explicadas anteriormente usando Apache Storm y Nvidia CUDA para extraer características de un ECG, se pudo concluir que la arquitectura propuesta puede usarse efectivamente para procesar señales de ECG y para algoritmos que requieren gran capacidad de computo y se deben efectuar para cantidades de datos muy grandes, Nvidia CUDA puede presentar una considerable mejora en el tiempo de ejecución frente a implementaciones secuenciales, haciendo que la arquitectura sea altamente eficiente. También se demostró al incrementar las instancias de las unidades de procesamiento y de las unidades receptoras de datos que esta arquitectura puede escalar fácilmente para atender una alta demanda de pacientes y balancearla entre todas las instancias disponibles.

Esta arquitectura puede ser utilizada para realizar un monitoreo del estado de salud de pacientes y extraer características e integrarlo con modelos de Machine Learning mas avanzados que el que se usó en este caso, como redes neuronales de Deep Learning para clasificación en distintas categorías de cardiopatías. Los hospitales e IPS pueden utilizar esto para agilizar el proceso de atención de pacientes y mejorar la calidad de vida de la sociedad.

6.2. Trabajo Futuro

La arquitectura que se implementó puede ser complementada con trabajo en el aspecto de seguridad de la información (como puede ser implementación de estándares propuestos por el OWASP, o por el HL7, o también "public physical unclonable functions"para

agregar seguridad al dispositivo de captura de datos), un lenguaje de dominio específico (DSL) para la composición de filtros sobre el modelo abstracto de la arquitectura que se planteó o la implementación de algoritmos de Deep Learning para mejorar el modulo de clasificación o la implementación de un sistema de escalado automático para la cantidad de instancias de las unidades de procesamiento, como se propone en [6].

Apéndice A

Sprints

Sprint 1:

- Investigación de marco teórico sobre el Corazón.
- Investigación de marco teórico sobre los electrocardiogramas.
- Investigación marco teórico sobre filtrado y extracción de características.

Sprint 2:

- Instalación y configuración de los servidores.
- Instalación y configuración de los servidores.
- Investigación tarjeta de vídeo Nvidia para integración CUDA - Apache Storm.
- Creación de Topología en Storm.

Sprint 3:

- Creación de simulador de pacientes
- Creación de visor local
- Pruebas de desempeño

Sprint 4:

- Investigación de marco teórico FFT

- Integración CUDA y Storm para FFT
- Pruebas de desempeño CUDA
- Desarrollo de artículo
- Desarrollo de Documentación
- Preparación para vitrina PGR1

Sprint 5:

- Investigación de bases de datos de ECG
- Investigación de estado del arte
- Desarrollo artículo
- Investigación de wavelets

Sprint 6:

- Extracción de datos de Base de datos Apnea
- Investigación y algoritmos Machine Learning para procesamiento de ECG
- Creación de modelo de Machine Learning
- Integración de modelo de Machine Learning con Storm

Sprint 7:

- Configuración de broker STOMP
- Creación del visor web
- Integración de la topología con el visor web

Sprint 8:

- Preparación para vitrina PGR2
- Desarrollo de Documentación
- Investigación de estado del arte
- Desarrollo de libro PGR

Apéndice B

Manuales

B.1. Requerimientos tecnicos del sistema

La topología de Storm realizada se trabajó en un cluster de 5 equipos con las siguientes especificaciones: CPU: Intel Core i7-3770 CPU @ 3.70GHz x 4 RAM: 7.7 GiB Disco Duro: 465 GiB Sistema Operativo: Linux Mint 18.1 Cinnamon 64-bit Dos de esos tienen una GPU NVIDIA quadro p400. Para describir los servicios instalados y configurados en cada equipo, serán referidos con los números del 1 al 5.

- 1: Storm nimbus
- 2: Zookeeper y Redis server
- 3 y 4: Storm supervisor (estos equipos tienen GPU)
- 5: VerneMQ y Apache Apollo

B.2. Manuales de instalación

B.2.1. Apache Storm

Apache Storm es un framework para el procesamiento de datos en tiempo real, utilizada para el procesamiento y análisis de datos. Es capaz de soportar cualquier lenguaje de programación. Es caracterizado por distribuir tareas mediante una estructura conocida como topologías, que no son más que grafos DAG, cuya principal característica es que es un grafo sin ciclos.

Los nodos de este grafo son clasificados en spout (nodos receptores), comúnmente representados por unos grifos y bolts (nodos de procesamiento) comúnmente representados con un rayo. Estos bolts y spouts comunican entre ellos constantemente una abstracción de Apache Storm conocida como Stream. Un spout es el encargado de recibir los datos desde las fuentes y pasarla a un o varios bolts para su procesamiento. Un bolt es una unidad de procesamiento de los datos que recibe desde un bolt o spout, generando unos datos procesados que pueden ser recibidos por otro bolt o simplemente ser el resultado final.

Una de las ventajas de Apache Storm es que podemos dividir las unidades de procesamiento es decir, si tenemos un bolt (nodo) en un un topología, que realiza un filtro para eliminar alguna anomalía de la señal podemos aumentar su número de instancias para disminuir cuellos de botella y así disminuir el tiempo en el cual la señal es tratada.

Para la configuración de Apache Storm en modo clúster se manejan tres roles principales, más adelante se detalla la configuración para cada uno de ellos:

Nimbus(Master): Es encargado de asignar las tareas, distribuir los datos y monitorear las posibles fallas de los supervisores.

Zookeeper (Worker node): Es el encargado de ayudar a el Nimbus a mantener una comunicación con los supervisores,compartir y mantener la información sincronizada. Nimbus depende del Zookeeper para monitorear el estado de los supervisores.

Supervisor (Worker node): Es el encargado de realizar las tarea asignadas según la topología configurada en un ambiente que se conocen como Worker process.

NOTA: El equipo en el que se trabajó la presente guía contaba con un sistema operativo LINUX MINT 18.1 Cinnamon 64-bit, un procesador INTEL CORE i7-3770@ 3.40GHz y una memoria RAM de 7.7 GB.

B.2.1.1. Requisitos previos

1. Instalación de Java

   a) Para la instalación de Java, se ejecutan los siguientes comandos:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

   b) Para verificar la versión de Java instalada, puede abrir la terminal y ejecutar el comando java -version como se muestra en la Fig. B.1.

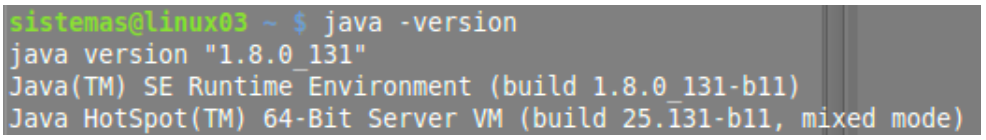


FIGURA B.1: Versión de Java

2. Instalación de Maven

   a) Para la instalación de Maven, se ejecutan los siguientes comandos:

```
sudo apt-get install maven
```

   b) Para verificar la correcta instalación de maven ejecutamos el comando mvn -version. El comando da el resultado mostrado en la Fig. B.2

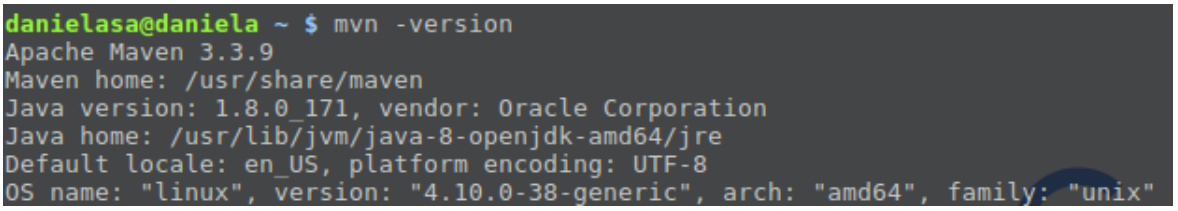


FIGURA B.2: Versión de Maven

B.2.1.2. Configuración Equipo Zookeeper

1. Para la instalación del equipo que jugará el papel de zookeeper. Primero actualizamos el sistema mediante el comando:

```
sudo apt-get -q -y update
```

2. Descargamos y descomprimimos los archivos del zookeeper de Apache Storm.

```
sudo wget http://apache.uniminuto.edu/zookeeper/zookeeper-3.4.12/zookeeper-3.4.12.tar.gz
```

3. Descomprimimos el archivo mediante el comando.

```
sudo tar -zxvf zookeeper-3.4.12.tar
```

4. Con el fin de mantener organizados nuestros archivos, renombramos y movemos la carpeta a la ruta /usr/local/zookeeper . (Este paso es opcional)

```
sudo mv zookeeper-3.4.12/ zookeeper/
sudo mv zookeeper /usr/local/zookeeper
```

5. Estando en la carpeta /usr/local/zookeeper creamos una carpeta de nombre data. En la cual nuestro zookeeper utilizará para guardar la información al momento de ejecutarse en algún clúster. Información como: logs, información de los supervisores, etc. Para la creación de la carpeta ejecutamos los siguientes comandos:

```
cd /usr/local/zookeeper
sudo mkdir data
```

6. En la carpeta conf encontramos el archivo zoo_sample.cfg realizamos una copia del mismo llamada zoo.cfg, mediante el comando:

```
cd /conf
sudo cp zoo_sample.cfg zoo.cfg
```

7. Hasta el momento los archivos en la carpeta zookeeper deben aparecer así, como se detalla en la Fig. B.3:

   - La carpeta data
   - El archivo de configuración zoo.cfg

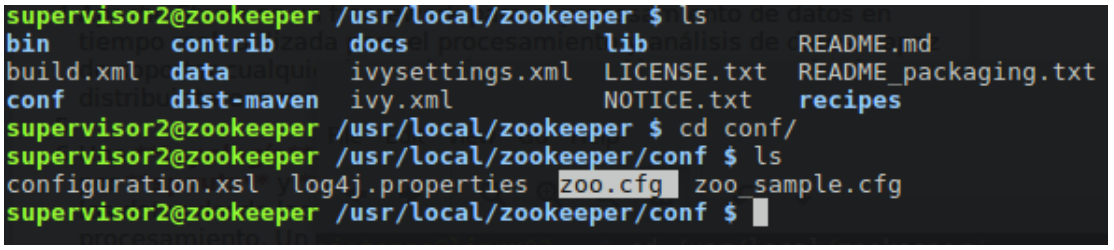


FIGURA B.3: Contenido carpeta Zookeeper

8. Se edita el archivo zoo.cfg en las siguientes líneas:

```
tickTime = 2000
dataDir = /usr/local/zookeeper/data
clientPort = 2181
initLimit = 5
syncLimit = 2
```

   Mediante el comando, quedando como se detalla en el Fig. B.4.

```
sudo nano zoo.cfg
```

9. Para iniciar el Zookeeper se ejecuta el script ./bin/zkServer.sh start, como se muestra en la Fig. B.5.

10. Para parar el Zookeeper se ejecuta el script ./bin/zkServer.sh stop, como se muestra en la Fig. B.6.

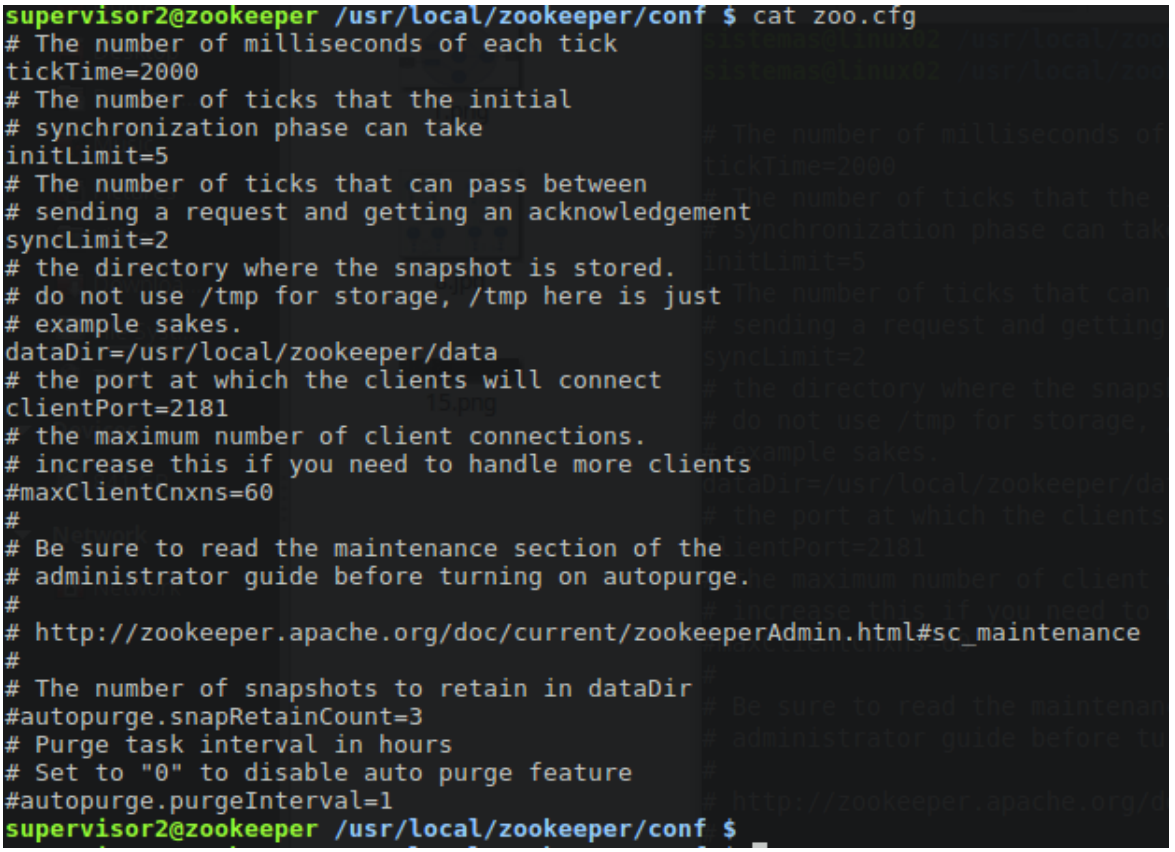


Figura B.4: Contenido archivo zoo.cfg

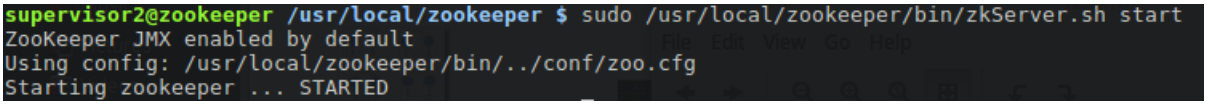


Figura B.5: Inicio de servicio Zookeeper

Figura B.6: Detener de servicio Zookeeper

B.2.1.3. Configuración Equipos Nimbus y Supervisores

1. Para la instalación del equipo que jugará el papel de Nimbus,el cual en nuestro caso será 10.8.0.27 en la VPN o 10.2.78.210 (Supervisor1). Primero instalamos y actualizamos el sistema mediante los comandos:

```
sudo apt-get -q -y install build-essential
sudo apt-get install python-minimal
sudo apt-get -q -y install uuid-dev
sudo apt-get -q -y install git
sudo apt-get -q -y install pkg-config libtool autoconf automake
sudo apt-get -q -y install unzip
sudo apt-get -q -y update
```

2. Descargamos e instalamos zeroMQ un sistema de mensajería de colas, mediante los comandos:

```
sudo wget http://download.zeromq.org/zeromq-2.1.7.tar.gz
sudo tar -xzf zeromq-2.1.7.tar.gz
```

Este paso puede ser opcional pero es recomendable para mantener el orden en nuestro equipo.

```
sudo mv zeromq-2.1.7/ zeromq
sudo mv zeromq /usr/local/zeromq
```

Nos dirigimos a la carpeta mediante el comando:

```
cd /usr/local/zeromq
```

Estando en la carpeta, instalamos zeromq mediante los comandos:

```
sudo ./autogen.sh
sudo ./configure
sudo make
sudo make install
```

[frame=single]

3. Estando en la carpeta /usr/local, descargamos e instalamos jzmq del repositorio de github (https://github.com/nathanmarz/jzmq).

```
cd /usr/local
sudo git clone https://github.com/nathanmarz/jzmq.git
```

Estando en la carpeta jzmq instalamos sus fuentes a través de los siguientes comandos para tener un resultado similar al de la Fig. B.7:

```
cd jzmq
sudo sed -i 's/classdist_noinst.stamp/classnoinst.stamp/g' src/Makefile.am
sudo ./autogen.sh
sudo ./configure
sudo make
sudo make install
```

4. Este paso puede omitirse si no se presenta ningún error. Solo si, se presenta algún problema al momento de hacer ./configure relacionado con el path de JAVA_HOME. Como se detalla en la Fig. B.8: Para conocer la ruta donde se encuentra

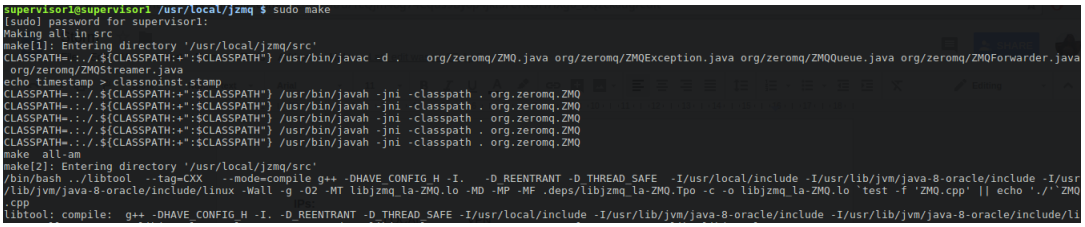


FIGURA B.7: Compilación de jzmq

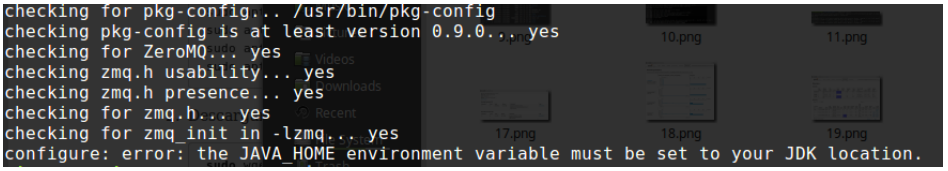


FIGURA B.8: Error en el ./configure

nuestro JAVA_HOME como en la Fig. B.9, se ejecuta el comando en una terminal:

```
echo $JAVA_HOME
```

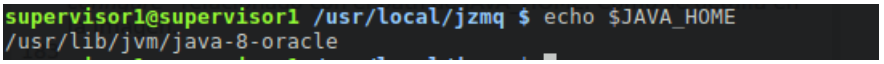


FIGURA B.9: Consulta de JAVA_HOME

Ya conociendo la ubicación de Java, se ejecuta el siguiente comando para editar el archivo configure:

```
sudo nano configure
```

Y se agrega las siguientes líneas, donde se revisa la conexión con el JDK. Para buscar en el editor de nano puede hacerse pulsando F6 ó CTRL + w.

```
export JAVA_HOME=/usr/local/java/jdk1.8.0\_77
echo $JAVA_HOME
```

Como se muestra en la Fig. B.10.

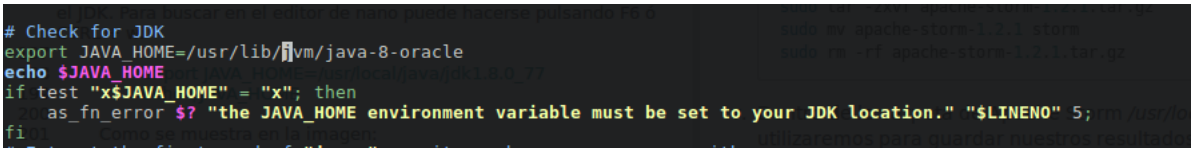


FIGURA B.10: Edición del archivo configure para agregar el JAVA_HOME

5. Desde la carpeta /usr/local/. Descargamos Apache Storm:

```
sudo wget http://apache.uniminuto.edu/storm/apache-storm-1.2.1/apache-storm-1.2.1.tar.gz
sudo tar -zxvf apache-storm-1.2.1.tar.gz
```

```
sudo mv apache-storm-1.2.1 storm
sudo rm -rf apache-storm-1.2.1.tar.gz
```

6. Dentro de la carpeta de Apache Storm /usr/local/storm creamos una carpeta llamada data, la cual utilizaremos para guardar nuestros resultados.

```
cd storm
sudo mkdir data
```

7. Entramos en la carpeta de configuración (conf). Dentro de está modificamos el archivo storm.yaml

```
cd conf/
sudo nano storm.yaml
```

8. Dentro del archivo agregamos nuestro zookeeper el cual será el equipo 10.2.67.2 (linux02). Se agregan las siguientes líneas como en la Fig. B.11:

```
nimbus.seeds: ["linux03.labinfo.is.escuelaing.edu.co"]
supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703
storm.local.dir: "/usr/local/storm/data"
storm.log.dir: "/usr/local/storm/logs"
topology.eventlogger.executors: 1
```

NOTA:

- Todo lo demás lo dejamos como comentario.
- Es recomendable trabajar con el host del equipo. Para saber el host del equipo lo podemos hacer mediante el comando: sudo hostname

9. Para la comunicación en los equipos del cluster es necesario editar el archivo /etc/hosts y agregar las direcciones y los hosts de los equipos que juegan el papel de zookeepers, del nimbus y de los supervisores, mediante el comando:

```
sudo nano /etc/hosts
```

En la Fig. B.12, se detalla cómo queda por ejemplo el archivo hosts del equipo zookeeper.

NOTA:

```
storm.zookeeper.servers:
     - "zookeeper"

nimbus.seeds: ["supervisor1"]

supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703

storm.local.dir: "/usr/local/storm/data"

storm.log.dir: "/usr/local/storm/logs"

topology.eventlogger.executors: 1

## UI CONFIG
ui.port: 8080
ui.actions.enabled: true
ui.pagination: 20

##Logs
logviewer.port: 8000
logviewer.max.sum.worker.logs.size.mb: 4096
logviewer.max.per.worker.logs.size.mb: 2048
logviewer.cleanup.age.mins: 10080

##Topology
topology.debug: true
```

FIGURA B.11: storm.yaml

```
127.0.0.1       localhost
127.0.1.1       zookeeper
10.2.78.212     supervisor3
10.2.78.211     zookeeper
10.2.78.210     supervisor1
10.2.78.213     supervisor4
10.2.78.214     supervisor5

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

FIGURA B.12: Archivo /etc/hosts del zookeeper

- Se deben actualizar todos los hosts de los equipos, con el fin de que se puedan comunicar.

10. Para iniciar nimbus ejecutamos el siguiente comando:

```
./bin/storm nimbus
```

Luego de ejecutar el comando el resultado es el detallado en la Fig. B.13.

```
^Csupervisor1@supervisor1 /usr/local/storm $ sudo ./bin/storm nimbus
[sudo] password for supervisor1:
Running: java -server -Ddaemon.name=nimbus -Dstorm.options= -Dstorm.home=/usr/local/storm -Dstorm.log.dir=/usr/local/storm/logs -Djava.library.path=/usr/local/lib:/opt/
local/lib:/usr/lib -Dstorm.conf.file= -cp /usr/local/storm/*:/usr/local/storm/lib/*:/usr/local/storm/extlib/*:/usr/local/storm/extlib-daemon/*:/usr/local/storm/conf -Xm
x1024m -Dlogfile.name=nimbus.log -DLog4jContextSelector=org.apache.logging.log4j.core.async.AsyncLoggerContextSelector -Dlog4j.configurationFile=/usr/local/storm/log4j2
/cluster.xml org.apache.storm.daemon.nimbus
```

FIGURA B.13: Inicio de proceso nimbus

11. Para iniciar el entorno gráfico de Apache Storm, ejecutamos el siguiente comando:

```
./bin/storm ui
```

Luego de ejecutar el comando el resultado es el detallado en la Fig. B.14.

FIGURA B.14: Inicio de storm UI

12. Para iniciar el servicio de supervisor en su respectivo equipo, en lugar de ejecutar el comando para el nimbus, ejecutamos el siguiente comando:

```
./bin/storm supervisor
```

Luego de ejecutar el comando el resultado es el detallado en la Fig. B.15.

FIGURA B.15: Inicio servicio supervisor

13. Luego de haber ejecutado el comando ./bin/storm ui podemos acceder a interfaz gráfica que nos proporciona Apache Storm mediante 10.8.0.27:8080 o en 10.2.78.210:8080 y se verá similar a com ose muestra en la Fig. B.16.

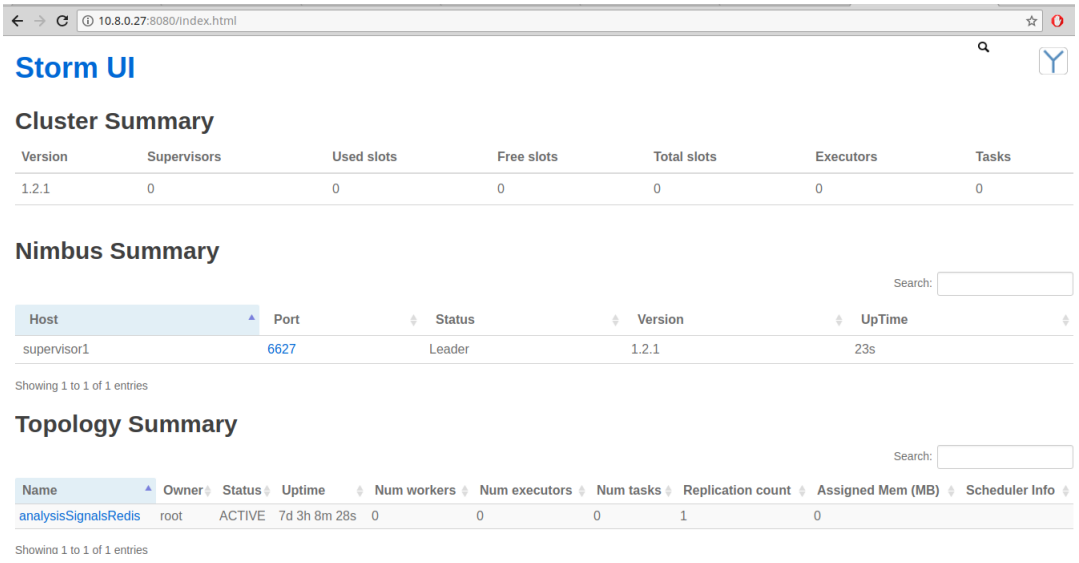


FIGURA B.16: Interfaz web de Apache Storm

NOTA:

- Primero se inicia el servicio de Zookeeper, luego el Nimbus y la interfaz Web. Después los supervisores de Apache Storm.

B.2.2. NVIDIA Cuda

A continuación se describirán los pasos a tener en cuenta antes de instalar CUDA en un equipo linux.

B.2.2.1. Pasos antes de la instalación

1. Verificar que se cuenta con una tarjeta compatible compatible Debemos ejecutar en la línea de comandos el siguiente comando

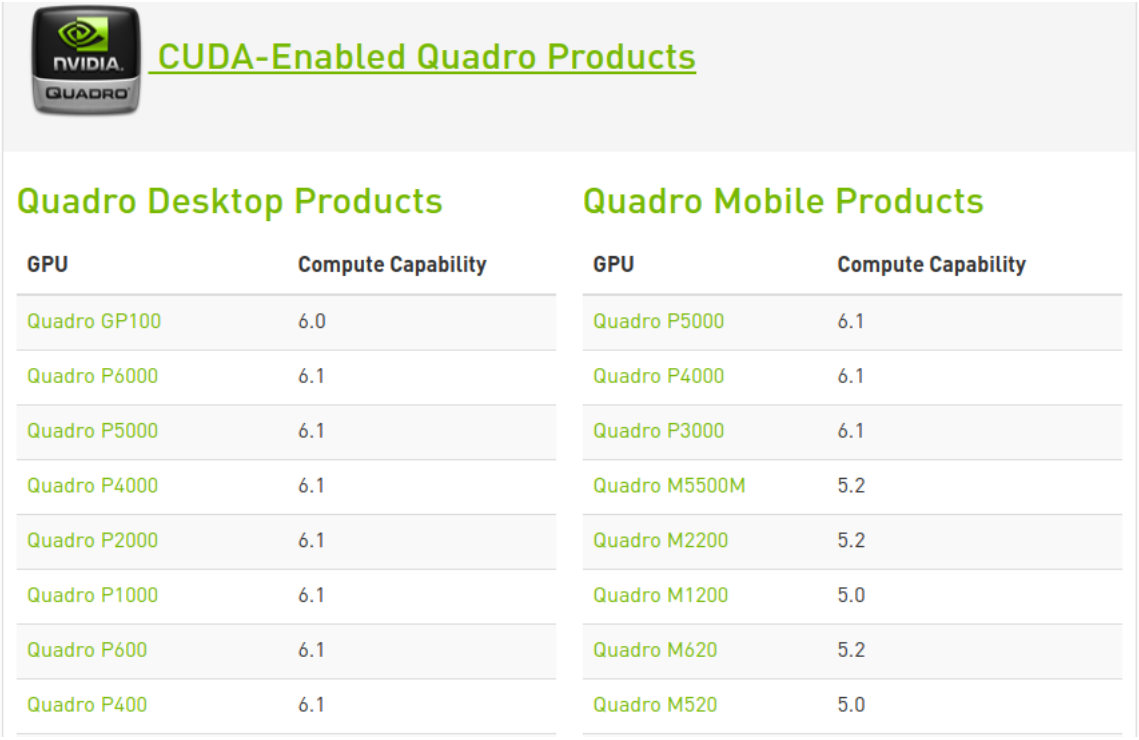
```
lspci | grep -i nvidia
```

La salida en este caso fue como se muestra en la Fig. B.17.

```
supervisor3@supervisor3 ~ $ lspci | grep -i nvidia
01:00.0 VGA compatible controller: NVIDIA Corporation Device 1cb3 (rev a1)
01:00.1 Audio device: NVIDIA Corporation Device 0fb9 (rev a1)
```

FIGURA B.17: Resultado lspci para buscar dispositivo conectado

Observamos en la página de NVIDIA si esta tarjeta gráfica es compatible con CUDA (en este caso el código 1cb3 es el código asignado a la NVIDIA Quadro P400) como se muestra en la Fig. B.18.



CUDA-Enabled Quadro Products

Quadro Desktop Products

GPU	Compute Capability
Quadro GP100	6.0
Quadro P6000	6.1
Quadro P5000	6.1
Quadro P4000	6.1
Quadro P2000	6.1
Quadro P1000	6.1
Quadro P600	6.1
Quadro P400	6.1

Quadro Mobile Products

GPU	Compute Capability
Quadro P5000	6.1
Quadro P4000	6.1
Quadro P3000	6.1
Quadro M5500M	5.2
Quadro M2200	5.2
Quadro M1200	5.0
Quadro M620	5.2
Quadro M520	5.0

FIGURA B.18: lista de dispositivos compatibles con CUDA

2. Verificar que el sistema operativo tenga soporte para CUDA Debemos ejecutar en la línea de comandos el siguiente comando:

```
uname -m && cat /etc/*release
```

La salida en este caso fue como se ve en la Fig. B.19.

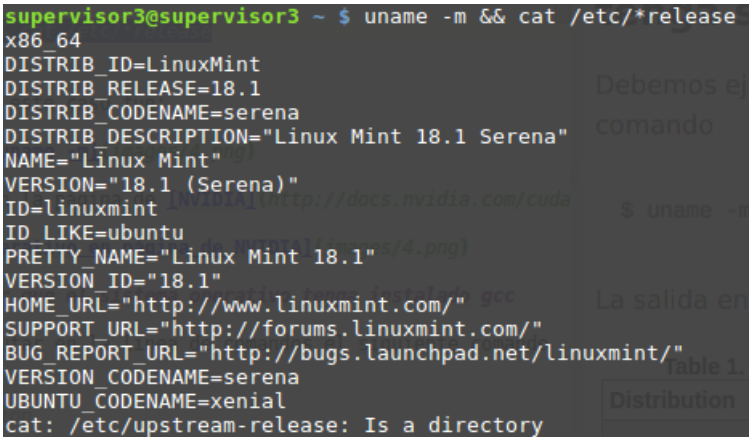


FIGURA B.19: Resultado de uname -m && cat /etc/*release

Observamos en la página de NVIDIA si este sistema operativo tiene soporte para CUDA, y se encuentra la tabla que se ve en la Fig. B.20 (en este caso linux mint es un derivado de ubuntu, así que si).

3. Verificar que el sistema operativo tenga instalado gcc Debemos ejecutar en la línea de comandos el siguiente comando:

```
gcc --version
```

La salida en este caso fue la que se ve en la Fig. B.21 lo que nos indica que el sistema cuenta con gcc 5.4.0.

4. Verificar que el sistema operativo tenga instalados los encabezados del kernel y paquetes de desarrollo correctos. Para conocer la versión de kernel que se está usando en el sistema se utiliza el comando

```
uname -r
```

En este caso para instalar los encabezados del kernel y los paquetes de desarrollo dado que la distribución de linux que se está usando es derivada de Ubuntu el comando es:

```
sudo apt-get install linux-headers-$(uname -r)
```

Table 1. Native Linux Distribution Support in CUDA 8.0

Distribution	Kernel	GCC	GLIBC	ICC	PGI	XLC	CLANG
x86_64							
RHEL 7.x	3.10	4.8.2	2.17	15 16	16.3+	NO	3.8+
RHEL 6.x	2.6.32	4.4.7	2.12				
CentOS 7.x	3.10	4.8.2	2.17				
CentOS 6.x	2.6.32	4.4.7	2.12				
Fedora 23	4.2.3	5.3.1	2.22				
OpenSUSE 13.2	3.16.6	4.8.3	2.19				
SLES 12	3.12.28	4.8.6	2.19				
SLES 11 SP4	3.0.101	4.3.4	2.11				
Ubuntu 16.04	4.4.0	5.3.1	2.23				
Ubuntu 14.04	3.13	4.8.2	2.19				
ARMv8 (aarch64)							
Ubuntu 14.04	3.13	4.8.2	2.19	NO	NO	NO	NO
POWER8(*)							
RHEL 7.x	3.10	4.8.2	2.17	NO	NO	13.1	NO
Ubuntu 16.04	4.4.0	5.3.1	2.23	NO	NO	13.1	NO

Table 2. Cross-build Environment Linux Distribution Support in CUDA 8.0

Host Distribution	Targeting Architectures (Linux)
x86_64	ARMv8 (aarch64)
Ubuntu 14.04	YES

FIGURA B.20: Sistemas linux soportados por Nvidia CUDA

```
supervisor3@supervisor3 ~ $ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

FIGURA B.21: Resultado gcc –version

Al ejecutar estos comandos obtenemos la salida que se muestra en la Fig. B.22 lo que nos indica que en este caso ya se encuentran instalados.

B.2.2.2. Instalación de CUDA

En este caso se seleccionó instalar desde el ejecutable de nvidia con el siguiente procedimiento:

1. Se descarga CUDA con el comando, en este caso la versión 9.1

```
wget https://developer.nvidia.com/compute/cuda/9.1/Prod/local_installers/cuda_9.1.85_387.26
```

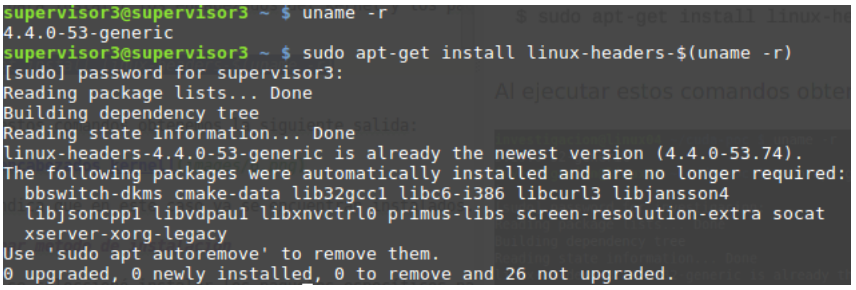


FIGURA B.22: Instalación de headers del kernel

2. Se descarga el driver de NVIDIA CUDA para linux, en este caso el driver 375

```
wget http://us.download.nvidia.com/XFree86/Linux-x86_64/375.20/NVIDIA-Linux-x86_64-375.20.r
```

3. Es necesario apagar el xserver si se encuentra corriendo

```
sudo service mdm stop
```

4. Instalar el driver de NVIDIA CUDA

```
sudo ./NVIDIA-Linux-x86_64-375.20.run
```

5. En este caso las opciones por defecto son suficientes para el funcionamiento correcto

6. Instalar CUDA

```
sudo ./cuda_9.1.85_387.26_linux
```

En este caso las opciones por defecto son suficientes para el funcionamiento correcto

B.2.2.3. Pasos luego de la instalación de CUDA

1. Configuración de ambiente Es útil para poder utilizar la misma configuración de ambiente con diferentes versiones no dejar las variables de ambiente fijas a la carpeta de versión, para esto si el instalador no lo hace, es recomendable crear un link a la carpeta de cuda por ejemplo:

```
sudo ln -s /usr/local/cuda /usr/local/cuda-9.1
```

   Para utilizar CUDA es necesario que se modifiquen las variables de ambiente PATH y LD_LIBRARY_PATH, lo que se hace con los siguientes comandos para la sesión actual:

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

Para dejar esta configuración de las variables como configuración por defecto de nuestro usuario en los posteriores inicios de sesión, colocamos estas líneas al final del archivo /.profile (o al final de /etc/profile si se desea que quede para todos los usuarios):

```
PATH="/usr/local/cuda/bin:$PATH"
LD_LIBRARY_PATH="/usr/local/cuda/lib64:$LD_LIBRARY_PATH"
```

2. Instalar ejemplos con permisos de escritura

   Para poder modificar, compilar y correr los ejemplos es necesario correr un script de instalación provisto por cuda con el siguiente comando

   ```
   cuda-install-samples-9.1.sh <directorio para ejemplos>
   ```

   Ahora dentro del directorio especificado se encuentra otra carpeta llamada NVIDIA_CUDA-9.1_Samples y contiene diferentes tipos de ejemplos que se podrán modificar, compilar y ejecutar.

3. Verificar instalación Para verificar la instalación de CUDA se procederá a compilar y ejecutar algunos de los programas de prueba

   a) Compilar los ejemplos Los ejemplos que se copiaron anteriormente se encuentran en forma de código fuente, para compilarlos es necesario ir hacia la carpeta en la cual se copiaron y acceder a NVIDIA_CUDA-9.1_Samples y ejecutar:

   ```
   make
   ```

   Este proceso puede tardar algunos minutos y durante este tiempo aparecerán en la consola varios Warnings, pero no deben ser motivo de preocupación

   b) Correr los binarios Se ejecutara el ejemplo deviceQuery que se encuentra en la ruta NVIDIA_CUDA-9.1_Samples/bin/x86_64/linux/release luego de compilar los ejemplos, se ejecuta con el comando:

   ```
   ./deviceQuery
   ```

   Al ejecutar el ejemplo nos muestra todas las especificaciones técnicas de nuestra tarjeta de vídeo, en este caso la salida es como se muestra en la Fig. B.23, Si se obtiene una salida similar a esta, cuda está funcionando correctamente.

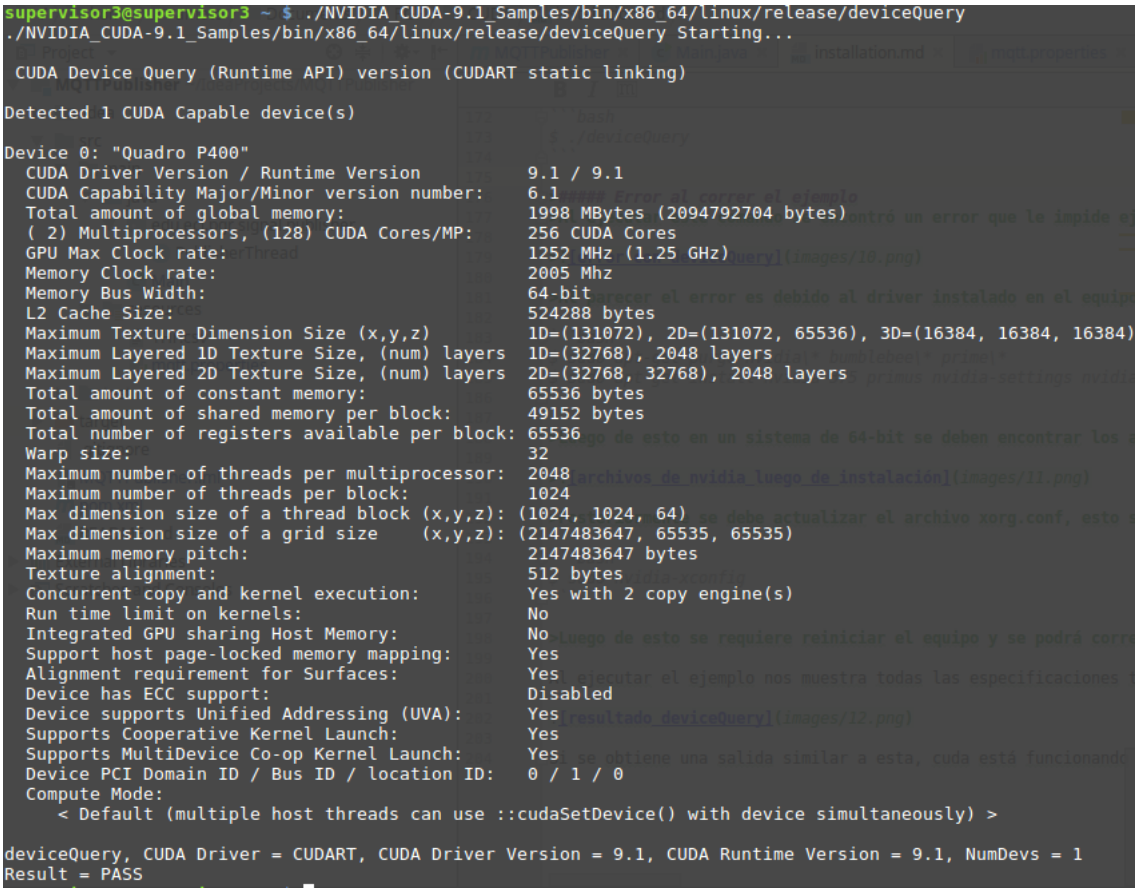


FIGURA B.23: deviceQuery

c) Error al correr el ejemplo Al ejecutar este comando se encontró un error que le impide ejecutar el programa, como se puede ver en la Fig. B.24.

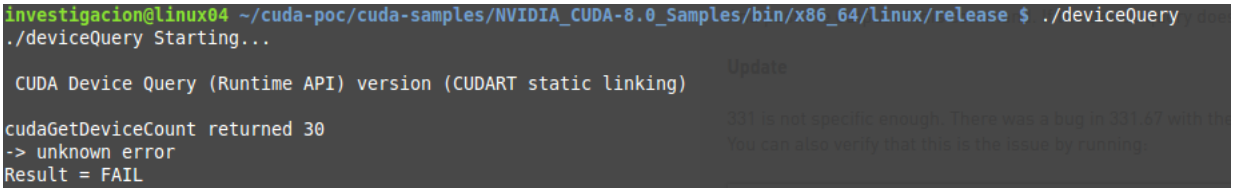


FIGURA B.24: Error al ejecutar el deviceQuery

Al parecer el error es debido al driver instalado en el equipo, este error se puede solucionar desinstalando todo el contenido de nvidia que se encuentre en el equipo, e instalando los drivers correctos, en este caso el driver es el 375 y se realiza con los siguientes comandos:

```
sudo apt-get purge nvidia\* bumblebee\* prime\*
sudo apt-get install nvidia-375 primus nvidia-settings nvidia-profiler nvidia-visual-profiler cuda
```

Luego de esto en un sistema de 64-bit se deben encontrar los archivos de NVIDIA en /usr/lib/ como se ve en la Fig. B.25.

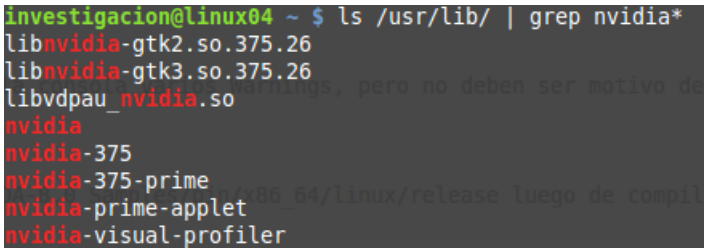


FIGURA B.25: Contenido de /usr/lib/ luego de instalar cuda

Posteriormente se debe actualizar el archivo xorg.conf, esto se hace ejecutando el siguiente comando:

```
sudo nvidia-xconfig
```

Luego de esto se requiere reiniciar el equipo y se podrá correr el ejemplo deviceQuery.

B.2.3. Redis

1. Para la instalación de redis, primero actualizamos el sistema e instalamos algunas dependencias mediante el comando:

```
sudo apt-get update
sudo apt-get install build-essential tcl
```

2. Se descarga de la página oficial redis y se descomprime

```
curl -O http://download.redis.io/redis-stable.tar.gz
tar xzvf redis-stable.tar.gz
```

3. Para mantener el orden de nuestro equipo movemos la carpeta descomprimida a /usr/local

```
sudo mv redis-stable /usr/local
```

4. Entramos en la carpeta de /usr/local/redis-stable e instalamos

```
cd /usr/local/redis-stable
sudo make
sudo make test
sudo make install
```

NOTA

- Ya ejecutado el make install podemos correr el servidor de redis con el comando:

```
redis-server
```

- Para ejecutar al cliente ejecutamos:

```
redis-cli
```

B.2.4. VerneMQ

Para este proyecto se utilizó VerneMQ como broker de MQTT porque este permite una fácil configuración para que múltiples clientes se conecten con el mismo id, así como también la opción de suscripción compartida que son las opciones que se encontraron útiles al momento de querer escalar la cantidad de instancias del spout.

Sitio oficial: https://vernemq.com/

B.2.4.1. Proceso de instalación

1. Descargar el .deb de la siguiente página: https://bintray.com/artifact/download/erlio/vernemq/de
   1_amd64.deb

```
wget https://bintray.com/artifact/download/erlio/vernemq/deb/xenial/vernemq_1.3.1-1_amd64.d
```

2. Se instala el .deb mediante el siguiente comando

```
sudo dpkg -i vernemq_1.3.1-1_amd64.deb
```

3. Se prueba la instalación mediante el comando

```
dpkg -s vernemq | grep Status
```

4. Debe retornar un mensaje parecido en la consola

```
Return Status: install ok installed
```

5. Se inicia el servicio mediante el comando

```
service vernemq start
```

B.2.4.2. Configuración del servicio

Para este ejercicio se configuró VerneMQ para que permitiera conecciones de diferentes clientes con el mismo id, adicionalmente, se desactivo la autenticación con contraseña, esto se realizó en el archivo de configuración adjunto (vernemq.conf). El archivo debe ser copiado a la carpeta /etc/vernemq/

B.2.5. Apache Apollo

Se seleccionó Apache Apollo para cumplir la función de broker para STOMP Sitio oficial: https://activemq.apache.org/apollo/

B.2.5.1. Proceso de instalación

1. Descargar apache apollo del repositorio de apache correspondiente con el comando:

```
wget http://apache.uniminuto.edu/activemq/activemq-apollo/1.7.1/apache-apollo-1.7.1-unix-di
```

2. Descomprimir el archivo descargado con el comando:

```
tar -zxvf apache-apollo-1.7.1-unix-distro.tar.gz
```

3. Crear una instancia del broker

```
cd /var/lib/
sudo [ruta_descarga_apollo]/apache-apollo-1.7.1/bin/apollo create [nombre_broker]
```

4. Se inicia el servicio mediante el comando

```
sudo /var/lib/[nombre_broker]/bin/apollo-broker run &
```

B.2.5.2. Configuración del servicio

El archivo de configuración de Apollo se utilizó en el archivo adjunto (apollo.xml), el archivo debe ser copiado a la carpeta /var/lib/[nombre_broker]/etc/

B.3. Manual de Usuario

Para un paciente, una vez se encuentra conectado el dispositivo de captura, solo necesita entrar al visor web diseñado para la arquitectura y allí podrá monitorear su estado de salud, para describir el uso de esta herramienta se cuenta con un vídeo de apoyo (https://www.youtube.com/watch?v=SSH8KQYCUQc), así como también las instrucciones que se describirán a continuación.

1. Acceder a la dirección del visor donde lo primero que se verá será el espacio para indicar la identificación del paciente a monitorear como se ve en la Fig. B.26.

FIGURA B.26: Primera vista del visor web

2. Si no se coloca ninguna identificación para el paciente se mostrará un error informando al respecto, como se ve en la Fig. B.27.

FIGURA B.27: Error cuando no se ingresa la identificación del paciente en el visor web

3. se debe colocar el número o código de identificación del paciente como se ve en la Fig. B.28 y oprimir el botón de Ingresar.

FIGURA B.28: Ingreso a la vista del paciente en el visor web

4. Una vez se ha ingresado en el visor se mostrarán en la parte superior los datos del paciente y abajo de estos, el espacio para graficar el ECG y los datos de ritmo cardíaco como se ve en la Fig. B.29.

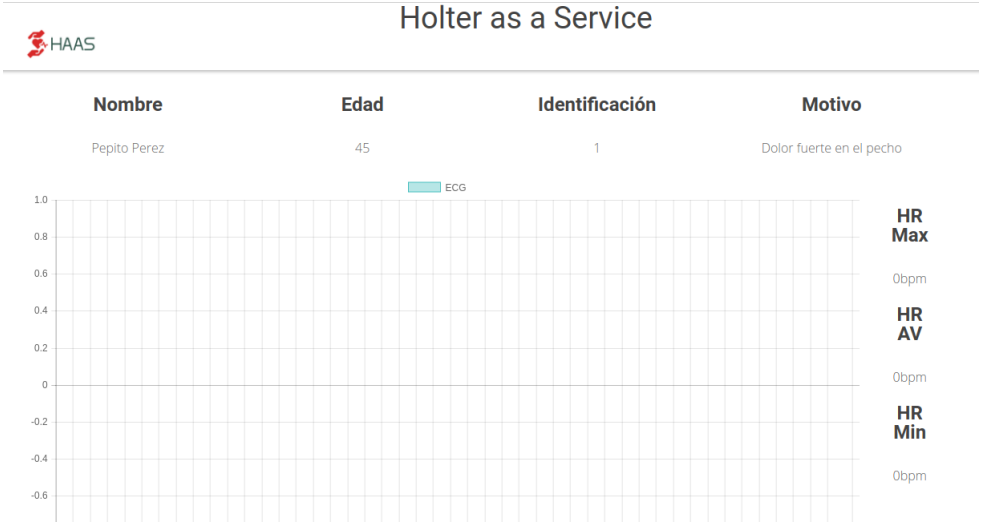


FIGURA B.29: Visor web una vez se ha ingresado en él

5. Cuando el dispositivo de captura esté emitiendo los datos se comenzará a ver la gráfica del ECG con un retraso de 5 segundos con respecto al dispositivo de captura, también se actualizarán los valores del ritmo cardíaco y la frecuencia de muestreo como se ve en la Fig. B.30.

FIGURA B.30: Visor web graficando un ECG

6. Debajo de la gráfica se encuentra una tabla que se actualizará en tiempo real con los valores de las características que se calcularon del ECG así como su clasificación (A si el paciente presenta Apnea o N si el ECG se ve normal) como se puede apreciar en la Fig. B.31.

Time	pNN50	NN50	pNN20	NN20	SDDNN	rMSSD	HR AV	HR MAX	HR MIN	Frec. Significativa	Clasificación
04-08-2018 21:29:17	40.0%	2	40.0%	2	67.3068	100.8646	37.0000bpm	75.6000bpm	1.2000bpm	34.7656Hz	A
04-08-2018 21:29:23	40.0%	2	40.0%	2	61.5370	92.2750	34.4000bpm	68.4000bpm	1.2000bpm	29.6875Hz	A
04-08-2018 21:29:29	60.0%	3	60.0%	3	46.0141	84.0040	41.9000bpm	51.0000bpm	49.8000bpm	35.9375Hz	N
04-08-2018 21:29:35	50.0%	2	50.0%	2	56.5862	98.0051	47.5200bpm	60.6000bpm	58.2000bpm	24.2188Hz	A
04-08-2018 21:29:41	71.428571428571143%	5	71.428571428571143%	5	37.8720	77.6041	35.0250bpm	47.4000bpm	1.2000bpm	30.8594Hz	N

FIGURA B.31: Tabla mostrando las características en el visor web

B.4. Manual Técnico

B.4.1. Guía para el desarrollo e integración de nuevos filtros básicos

En una topología en Apache Storm, los flujos de datos se conocen como Stream. Un Stream no es más que la secuencia de tuplas, las tuplas es la manera como se encapsula la información, una tupla puede ser de tipo: un String, double, int, un arreglo (Array) o un objeto. Las tuplas enviadas a través de los diferentes nodos de la topología requieren una confirmación por parte de los receptores de las tuplas a los emisores para eso existe una clase llamada OutputCollector que contiene dos métodos importantes uno de ack(Tuple input) y fail(Tupla input) muy parecido a los protocolos orientados a conexiones de la capa de transporte. Sí el mensaje es recibido y procesado con éxito el receptor hará ack, informando a su emisor, de lo contrario alerta el emisor haciendo fail de la tupla.

Los spouts pueden ser de tipo reliable capaz de retransmitir las tuplas, que han fallado en el envió o ser de tipo unreliable se olvidan de las tuplas una vez emitidas, no es necesario hacer un ack de parte del receptor para confirmar su envío. Existen muchas interfaces para la implementación de un Spout (Fig. B.32), todo dependiendo de lo que se requiere o necesita en la imagen anterior se muestran a grandes rasgos algunas.En la presente guía se implementará un BaseRichSpout que es uno de los más sencillos.En los cuales sólo es necesario implementar tres métodos:

- **public open(Map conf, TopologyContext context, SpoutOutputCollector collector)**: Esté método es invocado cuando se inicia la ejecución de la topología.
- **public void nextTuple(Tuple input)**: Lee, procesa o emite una tupla a los bolts.
- **public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer)**: Los Stream o flujos de información que emitirá el Spout a cualquiera de los nodos.

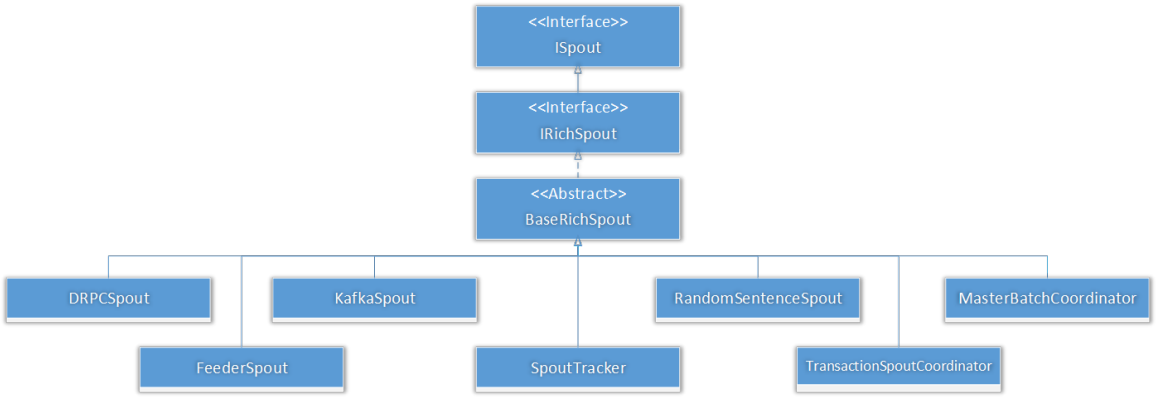


FIGURA B.32: Jerarquía de interfaces para spouts

Fuente: http://www.allprogrammingtutorials.com/tutorials/topologies-streams-spouts-and-bolts-in-storm.php

Los bolts estos son los que procesarán la información que contiene la tupla que ha recibido desde un spout o de otro bolt. Existen muchas interfaces para implementar un bolt, todo depende de que transformación o procedimiento se requiere. En la presente guía se implementará un BaseRichBolt que es uno de los más sencillos.En los cuales sólo es necesario implementar tres métodos **public void prepare(Map stormConf, TopologyContext context, OutputCollector collector), public void execute(Tuple input) y public void declareOutputFields(OutputFieldsDeclarer declarer)** de los cuales hablaremos más adelante en el desarrollo del ejemplo.

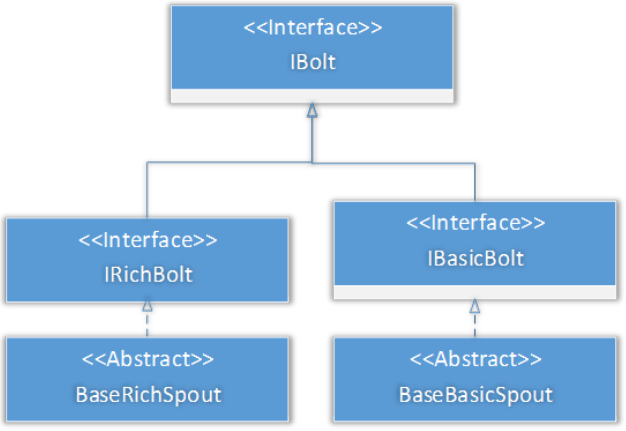


FIGURA B.33: Jerarquía de interfaces para bolts

Fuente: http://www.allprogrammingtutorials.com/tutorials/topologies-streams-spouts-and-bolts-in-storm.php

Las transmisión de las tuplas puede ser en de diferentes tipos, para mayor detalle revise esta página https://hadoopabcd.wordpress.com/2015/04/25/storm-real-time-data-processing/ y la documentación Apache Storm en http://storm.apache.org/releases/1.0.6/.

Existen cuatro maneras de manejar el flujo de los streams dentro una topología de Apache Storm (Fig. B.34), estos son:

- Shuffle grouping: Distribuye uniformemente al azar todas las tuplas con un número igual de tuplas procesado por cada tarea.
- Fields grouping: La tarea con el mismo Fields serán enviadas a un lugar especifico. Se definen así: builder.setBolt("FilterOne", new FilterOne()) .fieldsGrouping("frecuenciacardiaca", new Fields("Signal")) Es decir, el bolt FilterOne, recibirá stream con el nombre "frecuenciacardiaca".
- All grouping: Replica la tupla para todos.
- Global grouping: La tupla va a un solo lugar.

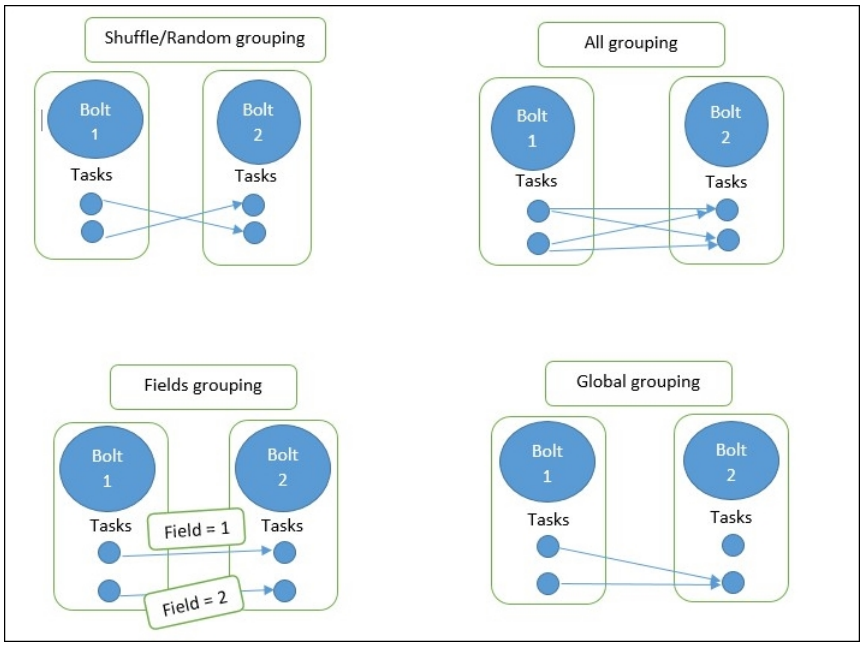


FIGURA B.34: Formas de envió de los streams

Fuente: https://dzone.com/articles/apache-storm-architecture

Para la creación de un Bolt en la topología primero se debe clonar el proyecto con las fuentes, del repositorio del proyecto bajo el nombre: scalable-ecg-storm-topology del nodo salud de la Escuela Colombiana de Ingeniería Julio Garavito. Una vez, clonado se crea una nueva clase, está debe extender BaseRichBolt, de la siguiente manera: public class NOMBRECLASE extends BaseRichBolt.

Es aconsejable tener definir los atributos al inicio de la clase:

```
private OutputCollector collector;
```

```
    private TopologyContext context;
    private Map conf;
    private static Logger LOG = LoggerFactory.getLogger(NOMBRECLASE.class);

    @Override
    public void prepare
    (Map stormConf, TopologyContext context, OutputCollector collector) {
        this.context = context;
        this.conf = stormConf;
        this.collector=collector;
    }
```

El método prepare, es el mismo para todos los Bolts.

```
    @Override
    public void execute(Tuple tuple) {
        try {
            collector.ack(tuple);
            /*
            INSERTE AQUI LO QUE HARA EL BOLT
            */
            collector.emit("STREAM",new Values(VALOR1, VLOR2));
            //Ejemplo, para el envio de una tupla.

            LOG.info("NOMBRECLASE._____."+tuple.getString(0));
            //Recordar escribir en los logs, para el seguimiento de errores.

        }catch (Exception e){
            e.printStackTrace();
            collector.fail(tuple);

            LOG.error("Error!"+e.getMessage()+" "+ NOMBRECLASE.class);
            //Ejemplo de reporte de error.
        }
    }
```

El método execute cambiá de acuerdo a la función que se quiera para el bolt.

```
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declareStream("FFTr", new Fields("ID", "signal"));
        //Ejemplo del envio de un Strem por Fieldsgrouping.
    }
```

En el Main de la topología cuando se cree el Bolts debe declararse así:

```
builder.setBolt("frecuencyCalculator",new SingleFrequencyCalculatorBolt(),1)
               .fieldsGrouping("fft","FFT",new Fields("ID"));
               //Este recibira los stream con el Field, FFT del Bolt fft.
```

Los Streams agrupados por con fieldsGrouping permiten mantener el orden de llegada de las tuplas y así no se desordena el orden de la señal, lo cuál es muy importante para su procesamiente.

Para que la información de un bolt pueda ser enviada al frontend, en este caso, el visor web, es necesario que esta información sea enviada por el protocolo STOMP, para lo que se desarrolló un bolt llamado PublishStompBolt, el cual en su constructor recibe el sufijo del tópico STOMP al que enviara la información, solo se necesita conectarlo a un stream por el cual se esté enviando una cadena y con esto el bolt enviara la cadena por el protocolo STOMP para que el visor cuando se suscriba la reciba y la pueda manejar.

B.4.2. Guía para el desarrollo e integración de filtros que hagan uso del API CUDA

B.4.2.1. Desarrollo de filtros y funciones que hagan uso de CUDA

Una vez se tenga el algoritmo para el filtro o la función que se desea desarrollar, es necesario pensar el algoritmo de forma paralela para sacar provecho del API CUDA. En este caso se utilizó el API de NVIDIA directamente en C, también existen librerías para hacer uso de este desde otros lenguajes (Python, Pascal, Java), pero el desempeño, uso o integración de estas con la arquitectura de Storm puede ser un obstáculo, así que acá se darán recomendaciones para este proceso teniendo el desarrollo de CUDA en C e integrandolo con la topología de Storm desarrollada en Java.

Es clave tener en cuenta los recursos de la GPU conocer los conceptos que se manejan en este modelo de programación como:

- Kernel: El kernel es una función que se ejecuta en la GPU, se declara un kernel escribiendo __global__ al comienzo de una declaración de una función.
- Thread (hilo): Un hilo en CUDA son solo la ejecución de un kernel en la GPU.
- Block (bloque): Un bloque en CUDA es un conjunto de hilos para la ejecución en la GPU.
- Memoria Compartida: Es una porción de memoria de rápido acceso a la que pueden acceder todos los hilos dentro de un bloque.
- CUDA CORE: Los núcleos CUDA son las ALU de las que dispone cada bloque, es decir que cada bloque, en un ciclo de reloj procesa tantos hilos al tiempo, como núcleos tenga.

Para conocer las especificaciones y limitaciones de nuestro dispositivo se puede utilizar el ejemplo deviceQuery de CUDA que nos entrega toda esta información, como se mostró en la Fig. B.23.

Las limitaciones que tenemos al utilizar CUDA y debemos tener en cuenta para trabajar y desarrollar el programa es que el número de bloques y el número de hilos por bloque están limitados.

Con este resultado del deviceQuery, vemos que poseemos 256 núcleos CUDA para cada bloque, y con el resultado de Max dimension size of a thread block vemos que el límite de bloques e hilos es de 1024 hilos por bloque y 1024 bloques por hilo y esos son los límites que se deben tener en cuenta al momento de llamar el kernel, pues al momento de exceder cualquiera de estos el programa no funcionará correctamente (si se utilizan más bloques de los permitidos el programa dirá que se demoró demasiado en arrancar y no funcionará, mientras que si se excede el número de hilos dirá que los parámetros son incorrectos y tampoco funcionará).

En relación a la eficiencia es importante tener en cuenta que para decidir el número de hilos que se utilicen se debe tener en cuenta el uso de memoria compartida para que se aproveche al máximo, y también el número de núcleos, pues entre más cercano sea el número de hilos a un múltiplo del número de núcleos para que no se desperdicien ciclos de cómputo con las operaciones.

En la página de NVIDIA (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html) se encuentra una guía detallada acerca de como funciona y como comenar a programar utilizando el API de CUDA

B.4.3. Integración de filtro o función desarrollada en CUDA con Apache Storm

Para hacer uso de una función desarrollada en C o C++ desde Java, es necesario que esta sea compilada en formato .so en linux o .dll en windows, esto significa que se creará una biblioteca dinámica.

1. En el código fuente es necesario que no se encuentre una función main, sino la función que se desea utilizar.
2. Instalar Cmake y make para compilar de forma más sencilla y realizar los enlaces con las bibliotecas externas que se utilicen.
3. Crear el archivo CMakeLists.txt en el que se deben listar los archivos a compilar y las bibliotecas que se desean enlazar, el archivo utilizado en este proyecto se encuentra adjunto.
4. Se ejecutan los siguientes comandos desde la carpeta donde están los fuentes y el CMakeLists.txt para crear la biblioteca dinámica (hay que notar que al momento

de la compilación, el compilador le agrega el prefijo “lib” al archivo y la extensión .so):

```
cmake  .
make
```

5. Si se hace uso de bibliotecas externas, es necesario que se encuentren en una de las rutas por defecto del sistema, para que la máquina virtual de Java las encuentre de forma sencilla, para este propósito, en los equipos que iban a realizar esta tarea se creó un link simbólico de la biblioteca que se quería utilizar en la carpeta /usr/local/lib, el link se creó con el siguiente comando:

```
sudo ln -s /usr/local/cuda/lib64/libcufft.so.8.0 /usr/local/lib/libcufft.so.8.0
```

6. Debido a que Storm va a funcionar de forma distribuida, es necesario que la biblioteca dinámica creada en el paso 4 se encuentre en la misma ubicación en todas las máquinas que vayan a hacer uso de esta (en este caso, las 2 máquinas que iban a funcionar como supervisores en las que también se realizó el paso 5).

7. Agregar la dependencia de JNA al proyecto de Java de la topología para poder realizar los llamados a bibliotecas dinámicas.

```
<dependency>
    <groupId>com.sun.jna</groupId>
    <artifactId>jna</artifactId>
    <version>3.0.9</version>
</dependency>
```

8. Declarar en cual carpeta se encuentran las bibliotecas dinámicas que se van a utilizar, en este caso es en la carpeta /usr/local, esto se realiza con la siguiente línea que se colocó en el método “prepare” del bolt que se encargaba de utilizar la biblioteca:

```
System.setProperty("jna.library.path","/usr/local");
```

9. Al momento de compilar la biblioteca dinámica, esta cambia el nombre de las funciones que contiene dentro del objeto compilado, así que es necesario escanear el resultado de la compilación con el comando nm de la siguiente manera:

```
nm libecg.so | grep proceed
```

En este caso se le dice a nm, que examine el archivo libecg.so, y con grep se filtran los resultados para ver el nombre con el que quedó la función que se busca, en este caso proceed y el resultado se muestra en la Fig. B.35.

```
investigacion@linux04 ~/}/ECG/src $ nm libecg.so | grep proceed
0000000000006ba1 T _Z7proceediPi
```

FIGURA B.35: Resultado nm

10. Se crea una interface en java que se comportará como si fuera la biblioteca dinámica que se creó, como se menciona en el paso 4, el compilador agrega el prefijo "lib" y la extensión .so, pero para este paso, en el momento de realizar el Native.loadLibrary es necesario poner el nombre original, sin prefijo ni extensión (en este caso el nombre original era cufftecg, y el nombre del archivo era libcufftecg.so).

```
import com.sun.jna.Library;
import com.sun.jna.Native;
import com.sun.jna.Pointer;
import com.sun.jna.ptr.DoubleByReference;

public interface ECGCudaLibrary extends Library {
   ECGCudaLibrary INSTANCE = (ECGCudaLibrary) Native.loadLibrary("cufftecg",ECGCudaLibrary.
   DoubleByReference _Z5fftcuPdi (Pointer datain, int N);
}
```

11. Es importante tener en cuenta que cuando se trabaja con estructuras de datos como arreglos en C, este maneja los apuntadores distinto a como lo hace Java, por lo que no hay un mapeo directo de los tipos de datos, en este caso el mapeo fue necesario hacerlo de la siguiente manera para poder enviar como parámetro el apuntador de un arreglo y recibir como resultado el arreglo que representaba el apuntador retornado:

```
//Creacion del apuntador al arreglo de tamano datain.length que se envia como parametro
Pointer pointer = new Memory(datain.length * Native.getNativeSize(Double.TYPE));
//Copia de la informacion del arreglo original al apuntador
for (int i=0; i<datain.length;i++){
   pointer.setDouble(i*Native.getNativeSize(Double.TYPE), datain[i]);
}
//Llamado a la funcion de la biblioteca dinamica
DoubleByReference res= ECGCudaLibrary.INSTANCE._Z5fftcuPdi(pointer,datain.length);
//Extraccion del arreglo con tamano datain.length/2+1 representado por el apuntador
double[] ans=res.getPointer().getDoubleArray(0,datain.length/2+1);
```

En la página https://www.eshayne.com/jnaex/index.html se encuentran ejemplos que pueden ayudar a determinar la forma como se deben mapear los apuntadores que se retornan desde C

Bibliografía

[1] WH World Health Organization, World Health Organization, et al. The top 10 causes of death, 2018.

[2] Reporte de enfermedades cardiovasculares en colombia. URL https://www.minsalud.gov.co/salud/publica/PET/Paginas/Enfermedades-cardiovasculares.aspx.

[3] Reporte de nacimientos y defunciones 2017 dane. URL https://www.dane.gov.co/files/investigaciones/poblacion/pre_estadisticasvitales_2018pre-29-junio-2018.pdf.

[4] Las 10 principales causas de defunción oms, . URL http://www.who.int/es/news-room/fact-sheets/detail/the-top-10-causes-of-death.

[5] Top 10 causes of death who, . URL http://www.who.int/es/news-room/fact-sheets/detail/the-top-10-causes-of-death.

[6] Yu-Jung Ko, Hui-Ming Huang, Wei-Han Hsing, Jerry Chou, Hung-Chih Chiu, and Hsi-Pin Ma. A patient-centered medical environment with wearable sensors and cloud monitoring. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, pages 628–633. IEEE, 2015.

[7] Van-Dai Ta, Chuan-Ming Liu, and Goodwill Wandile Nkabinde. Big data stream computing in healthcare real-time analytics. In *Cloud Computing and Big Data Analysis (ICCCBDA), 2016 IEEE International Conference on*, pages 37–42. IEEE, 2016.

[8] Min Chen, Yujun Ma, Jeungeun Song, Chin-Feng Lai, and Bin Hu. Smart clothing: Connecting human with clouds and big data for sustainable health monitoring. *Mobile Networks and Applications*, 21(5):825–845, Oct 2016. ISSN 1572-8153. doi: 10.1007/s11036-016-0745-1. URL https://doi.org/10.1007/s11036-016-0745-1.

[9] Y. J. Ko, H. M. Huang, W. H. Hsing, J. Chou, H. C. Chiu, and H. P. Ma. A patient-centered medical environment with wearable sensors and cloud monitoring.

In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 628–633, Dec 2015. doi: 10.1109/WF-IoT.2015.7389127.

[10] Yongwoo Cho, Heonshik Shin, and Kyungtae Kang. Scalable coding and prioritized transmission of ecg for low-latency cardiac monitoring over cellular m2m networks. *IEEE Access*, 6:8189–8200, 2018.

[11] Ruben Braojos, Daniele Bortolotti, Andrea Bartolini, Giovanni Ansaloni, Luca Benini, and David Atienza. A synchronization-based hybrid-memory multi-core architecture for energy-efficient biomedical signal processing. *IEEE Transactions on Computers*, 66(4):575–585, 2017.

[12] N Yaakob, R Badlishah, A Amir, Siti Asilah binti Yah, et al. On the effectiveness of congestion control mechanisms for remote healthcare monitoring system in iot environment—a review. In *Electronic Design (ICED), 2016 3rd International Conference on*, pages 348–353. IEEE, 2016.

[13] Min Chen, Yujun Ma, Yong Li, Di Wu, Yin Zhang, and Chan-Hyun Youn. Wearable 2.0: Enabling human-cloud integration in next generation healthcare systems. *IEEE Communications Magazine*, 55(1):54–61, 2017.

[14] Junaid Mohammed, Chung-Horng Lung, Adrian Ocneanu, Abhinav Thakral, Colin Jones, and Andy Adler. Internet of things: Remote patient monitoring using web services and cloud computing. In *Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing (CPSCom), IEEE*, pages 256–263. IEEE, 2014.

[15] ST Aarthy and S Kolangiammal. Signal monitoring in a telemedicine system for emergency medical services. In *Smart Computing and Informatics*, pages 343–351. Springer, 2018.

[16] DuyHoa Ngo and Bharadwaj Veeravalli. Design of a real-time morphology-based anomaly detection method from ecg streams. In *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, pages 829–836. IEEE, 2015.