Universidad de Alcalá

Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo de Fin de Grado

Software empotrado de control y gestión de un monitor de neutrones

Autor: Hristo Ivanov Ivanov

Tutor: Óscar García Población

TRIBUNAL:

Presidente: _____

Vocal 1: _____

Vocal 2: _____

Fecha: _____ **Calificación:** _____

Agradecimientos

Quiero expresar mi gratitud hacia mi tutor por todo lo que me ha enseñado a lo largo de este trabajo.

Un sincero agradecimiento al equipo de CaLMa por todos los conocimiento que me habéis ofrecido.

Debo un especial reconocimiento al Parque Científico y Tecnológico de Guadalajara (GuadaLab) por la acogida, apoyo y medios que me han ofrecido.

Gracias Sergio Valeriano, Javier Mayoral, Elena Iauncu y Javier López por ayudarme con la elaboración de este documento.

Todo esto nunca hubiera sido posible sin el apoyo incondicional de mi familia. Esto es también vuestro premio.

Índice general

Índice de figuras

Capítulo 1

Introducción

El propósito de este documento es describir el *Trabajo de Fin de Grado* realizado por el autor. Este es parte del desarrollo de un nuevo sistema de adquisición de datos para un monitor de neutrones, instrumento que permite medir los niveles de radiación cósmica. Los objetivos del trabajo son la realización del software encargado de gestionar la adquisición de datos y el desarrollo de una herramienta Web dirigida a la gestión del sistema.

Empezamos este capítulo haciendo una breve introducción a los conceptos teóricos y el entorno en el que se desarrolla este trabajo. Continuamos estableciendo los objetivos y el diseño preliminar. Finalmente aclaramos algunas peculiaridades referentes al proceso de adquisición de datos. El segundo capítulo describe el entorno hardware en el que se desarrolla el software de adquisición. Los siguientes capítulos describen en detalle el trabajo realizado. El primero de estos es dedicado al software de adquisición. Debido a la separación impuesta por el modelo *Cliente-Servidor* se utilizarán dos capítulos para describir la herramienta Web, el primero dedicado al *Servidor* y el segundo al *Cliente*. El último capítulo de este documento recoge las conclusiones obtenidas del trabajo y la visión futura de este. El documento también incluye un apéndice, en este son descritos los pasos a seguir para implantar el software de adquisición y la herramienta Web. En este apéndice también son reflejados los aspectos más importantes del proceso de implementación. Finalmente el documento incluye una lista de referencias bibliográficas a las fuentes de información utilizadas en el trabajo.

1.1. Introducción a los rayos cósmicos

A principios del siglo *XX* se descubrió la existencia de los rayos cósmicos. Estos son partículas subatómicas provenientes del espacio exterior. Estas partículas, en su mayoría protones y núcleos de Helio, son muy energéticas debido a su gran velocidad. El origen de estas no está muy claro, pero sabemos que proceden del espacio exterior. Raras veces la actividad solar puede producir partículas tan energéticas.

Muchas de estas partículas inciden en la atmósfera terrestre. En las capas altas de la atmósfera se producen las primeras interacciones, estas partículas colisionan con las partículas que forman la atmósfera. Esta colisión es muy violenta y causa la división de las partículas originales en partículas secundarias. Estas a su vez pueden colisionar con otras partículas de la atmósfera para así formar aún más partículas secundarias. Vemos como una sola partícula proveniente del espacio produce el fenómeno denominado *cascadas atmosféricas*. Como es de esperar con cada choque consecutivo se pierde parte de la energía. Normalmente las partículas secundarias que alcanzan la superficie

terrestre tan solo tienen una pequeña fracción de la energía inicial. Si una partícula no posee la energía suficiente la cascada que es originada no se propaga hasta la superficie terrestre.

Como hemos dicho la mayor parte de la radiación cósmica proviene de fuera de nuestro sistema solar, pero está fuertemente relacionada con los ciclos solares. Los ciclos solares de 11 años aproximadamente afectan la actividad solar pasando por un mínimo y un máximo, donde los cambios son apreciables en la luminosidad y el campo magnético. Es este segundo, el campo magnético solar, el que afecta a la llegada de radiación cósmica a la Tierra. Al ser mayormente partículas con carga eléctrica en la presencia de un fuerte campo magnético estas son desviadas. A continuación detallamos los sucesos más comunes que pueden ser observados indirectamente a consecuencia de estudiar la cantidad de radiación cósmica.

- Ciclo solar. Como hemos explicado existe una fuerte relación entre la cantidad de radiación cósmica y la actividad solar. La radiación cósmica es un buen indicador de la actividad solar donde la relación es inversa. Menos radiación generalmente significa una actividad solar elevada.

- *Forbush decrease*[13]. Estos sucesos consisten en un descenso rápido de los niveles de radiación cósmica medida en la Tierra. Estos descensos son consecuencia de CME's(Eyección de masa coronal). La materia expulsada en un CME al ser en su mayoría plasma, extiende e intensifica el campo magnético solar. Como ya hemos explicado el aumento del campo magnético solar conlleva al descenso de radiación cósmica.

- *Ground level enhancements*(GLE). Eventualmente la actividad solar es tan elevada que el Sol es capaz de emitir partículas muy energéticas. Estas partículas son a veces tan energéticas que pueden generar cascadas atmosféricas que alcanzan la superficie terrestre. Estos sucesos son muy raros, entre 10 y 15 por década. A pesar de ser muy raros estos pueden tener un gran impacto en nuestras vidas cotidianas, pueden afectar el funcionamiento de la electrónica sensible que está en orbita e incluso la que está en tierra.

1.2. Monitor de neutrones

Un monitor de neutrones es una estación terrestre que monitoriza la llegada de partículas extraterrestres de forma indirecta a partir de las cascadas atmosféricas. Este está compuesto por cuatro capas especialmente diseñadas para capturar las partículas secundarias producidas en las cascadas atmosféricas. A continuación procedemos a explicar estas cuatro capas.

- **Reflector**. La primera capa, la exterior, consiste en un escudo reflector que tan solo deja pasar las partículas con energía alta. De esta manera todas las partículas generadas por el entorno inmediato que tienen baja energía rebotan y no influyen en la medición.

- **Productor**. Esta capa compuesta generalmente de material denso tiene como objetivo conseguir algo parecido a las cascadas atmosféricas. La idea es tener un material denso para que aumente la probabilidad de que las partículas secundarias impacten con las partículas del material y como resultado se produzcan aún más partículas. A las partículas generadas en esta capa se les da el nombre de

neutrones de evaporación. Son estas partículas las que finalmente serán medidas por el instrumento, también son las que le dan nombre. Los neutrones producidos tienen menos energía, por lo que son más fáciles de medir.

- **Moderador**. A pesar de que las partículas que tenemos a este nivel tienen tan solo una fracción de la energía original, estas aún siguen siendo demasiado energéticas para ser capturadas. Esta capa tiene como objetivo ralentizar, disminuir la energía, de las partículas para así poder capturarlas.

- **Contador**. Un contador proporcional o tubo contador generalmente está relleno de gas con propiedades específicas. Cuando el gas interacciona con los neutrones de evaporación este es ionizado, en el proceso son liberados electrones. Debido al campo eléctrico que atraviesa constantemente el gas estos electrones son acelerados hacia el ánodo, hilo conductor que atraviesa el tubo contador en su centro. Conforme los electrones ganan energía, estos pueden producir ionizaciones secundarias. El número total de electrones que llegan al ánodo se mantiene, sin embargo, proporcional a la energía de la partícula inicial. Al llegar al ánodo estos electrones producen una corriente eléctrica.

Los sistemas de adquisición están diseñados para recoger estas pequeñas corrientes eléctricas y medirlas. Tradicionalmente la medida que se realiza son eventos por minuto, las señales son capturadas, amplificadas y registradas en un contador que se reinicia cada minuto. A lo largo de este trabajo muchas veces nos referiremos a esta medición de eventos por minuto con el nombre de *cuentas*.

1.3. NMDB

NMDB[32] o *Neutron Monitor Database* es una red mundial de monitores de neutrones. Antes de proceder a hablar sobre la red en concreto expondremos las ventajas y objetivos de una red como esta.

- Espectro de energías. Al igual que el Sol, la Tierra tiene campo magnético. Este campo magnético repele con mayor fuerza en las regiones ecuatoriales que en los polos. Esto implica que solo las partículas más energéticas son perceptibles en las zonas ecuatoriales, mientras que en los polos las partículas no necesitan ser tan energéticas para alcanzar la superficie. Combinando datos de estaciones que se encuentran a diferentes latitudes podemos construir espectrogramas basados en la energía de las partículas.

- Anisotropía. Tener estaciones en diferentes lugares del globo terráqueo implica estar orientado en diferentes direcciones del espacio. Esto implica poder realizar estudios sobre la procedencia de eventos.

- Redundancia. Tener muchas estaciones implica detectar el mismo evento en más de una estación. Esto permite comparar los datos entre estaciones y descartar fluctuaciones grandes, rápidas y asiladas en una sola estación.

- Cooperación. Estar en una red implica mejorar la comunicación entre las diferentes estaciones. De esta manera los resultados son mejores y el avance más rápido.

Como ya hemos comentado NMDB es una red mundial, impulsada por la Comisión Europea. Actualmente la red supera las veinte estaciones y proporciona datos en tiempo real con resolución de un minuto. Los formatos de los datos están estandarizados entre las diferentes estaciones, esto ayuda al análisis científico de estos. Los datos en tiempo real son utilizados para la elaboración de un sistema de alarma GLE[33]. Como hemos explicado un GLE fuerte puede tener un gran impacto negativo en nuestras vidas. Es beneficioso poder detectar estos eventos lo antes posible y este es uno de los objetivos principales del NMDB.

1.4. CaLMa

CaLMa[34] o *Castilla la Mancha Neutron Monitor* es el primer y único monitor de neutrones en España. Este forma parte del NMDB, el equipo técnico responsable de la estación está profundamente implicado en desarrollar sistemas y herramientas que mejoran la red. Un ejemplo es el sistema de adquisición implantado en la estación, también implantado en otras estaciones de la red. La estación empezó a operar de forma plena en diciembre de 2012 y desde entonces lleva haciéndolo ininterrumpidamente con pequeñas excepciones. Desde su puesta en marcha la estación ha registrado 18 Forbush decreases. Desafortunadamente aún no ha habido ningún GLE que detectar, aunque este tendría que ser muy energético para ser detectado en una estación con tan poca latitud. A continuación procedemos a hablar más a fondo del estado actual de CaLMa, hablaremos del sistema de adquisición, base de datos, herramientas y técnicas que son utilizadas.

1.4.1. Sistema de adquisición

Como hemos mencionado el sistema de adquisición que está implantado en CaLMa es producto del propio equipo[36]. Este está basado en un sistema empotrado Linux. Si las señales generadas por los tubos contadores superan un determinado umbral, estas son covertidas en pulsos digitales por los amplificadores. A continuación, un circuito de adaptación se encarga de elevar los niveles de tensión de los pulsos digitales a niveles *TTL* y de adaptar la impedancia de la línea de comunicaciones si el amplificador así lo requiere. Seguidamente los pulsos son procesados por una FPGA que es la encargada de medir los eventos por minuto de los 18 canales. Aunque la estación tan solo tiene 15 tubos contadores el sistema está diseñado para soportar 18, este número es un estándar histórico. El software que se ejecuta en el sistema Linux tiene como tarea comunicarse con la FPGA, su labor es recoger las *cuentas* de cada minuto y guardar estas en una base de datos. El software hace uso del *Network Time Protocol* para asegurar la correcta sincronización temporal entre estaciones, esta sincronización permite contrastar los datos de diferentes estaciones. En la figura 1.1 podemos ver un diagrama de bloques que representa el sistema de adquisición.

Actualmente el equipo de CaLMa está desarrollando un nuevo sistema de adquisición de datos. El esquema presentado en la figura 1.1 es aplicable al nuevo sistema. Las diferencias entre los dos sistemas de adquisición radican en los componentes. La primera de estas está en los amplificadores. Los utilizados en el sistema actual generan pulsos fijos, mientras que los amplificadores utilizados en este nuevo sistema de adquisición generan pulsos cuyo ancho es proporcional a la energia de la señal original. El fin es medir la energía de las señales generadas por los tubos contadores, esta magnitud es de gran interés técnico.

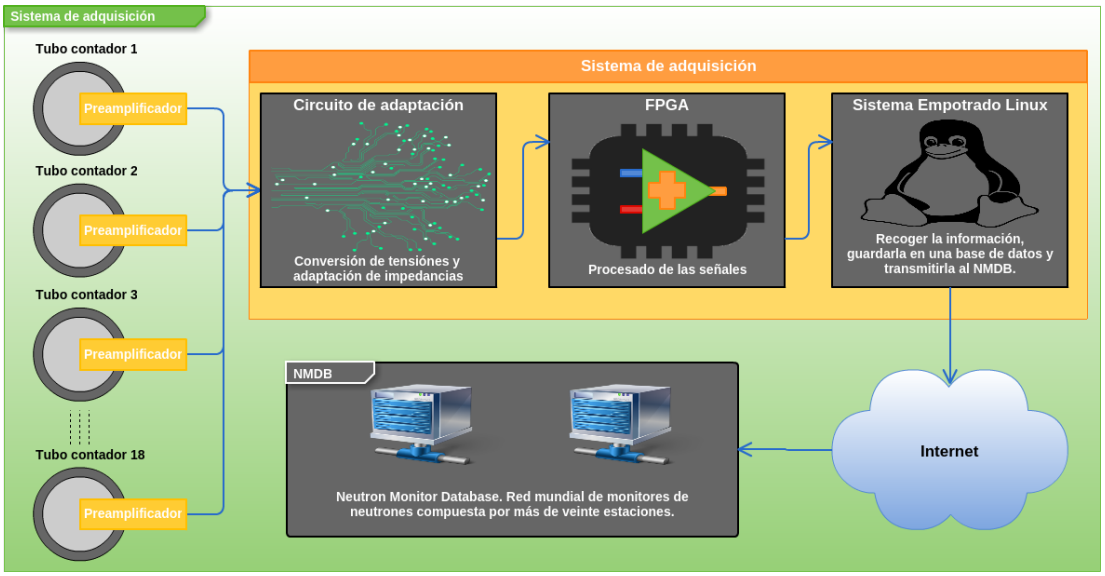

Figura 1.1: Sistema de adquisición.

La segunda diferencia está en el circuito de adaptación. Este cambia a fin de soportar los nuevos amplificadores que acabamos de explicar. La compatibilidad con los amplificadores utilizados en el sistema de adquisición actual también se mantiene. Esto da mucha flexibilidad a la estación, permitiéndole incorporar diferentes amplificadores a la vez.

La siguiente diferencia está en la FPGA, pero sobre todo en el *IP core* integrado en cada uno de los dos sistemas. Recordamos que en el sistema de adquisición actual este tan solo registra los eventos por minuto para los 18 canales. El nuevo *IP core* debe realizar la misma tarea, pero aparte debe poder medir el ancho de los pulsos generados por los amplificadores. Los dos sistemas de adquisición utilizan un puerto serie para la comunicación entre software y FPGA. En el nuevo sistema este puerto opera a una velocidad mayor a fin de poder transmitir toda la información extra.

El nuevo sistema también está pensado para ofrecer una mayor compatibilidad. El fin es poder implantar este en diferentes estaciones de forma fácil y rápida. El hardware y software deben ser diseñados con este requisito en mente.

1.4.2. Bases de datos

La base de datos que genera CaLMa está diseñada de acuerdo con el estándar impuesto por NMDB. Los eventos por minuto de los 18 canales se guardan en la tabla `binTable`. En una segunda tabla, `CALM_ori`, se guarda el valor global de la estación, calculado a partir de los datos de todos los tubos. Junto a este valor global también se guardan las correcciones que se realizan sobre este. El valor global y las correcciones se explican en la última sección de este capítulo. Los datos de esta segunda tabla son revisados por un operador humano y finalmente guardados en una tercera tabla cuyo nombre es `CALM_rev`. Los datos de estas dos últimas tablas se envian al NMDB. Este envío es realizado por un proceso que se ejecuta cada minuto. El envío consiste en calcular las diferencias entre los dos conjuntos de datos y tan solo enviar estas.

1.4.3. Herramientas y técnicas

El equipo de CaLMa hace uso de herramientas que les ayudan a analizar la información desde el punto de vista científico. Actualmente no existe ninguna herramienta que proporcione información sobre el estado técnico de la estación, todas las labores relacionadas con el mantenimiento son realizadas de forma manual.

1.5. Objetivos

El objetivo de este proyecto es desarrollar el software para el nuevo sistema de adquisición de datos y también desarrollar una herramienta que permite operar y monitorizar el estado de la estación. A continuación procedemos a hacer una descripción más detallada de los objetivos de este trabajo.

1.5.1. Software de adquisición

Tal y como hemos explicado en secciones anteriores el equipo de CaLMa está desarrollando un nuevo sistema de adquisición. Actualmente la mayoría de módulos hardware incluyendo la FPGA están listos. El propósito de este trabajo es realizar el software de adquisición. A continuación exponemos algunos de los requisitos más relevantes.

- El software debe ser capaz de realizar una correcta comunicación con la FPGA. Esto implica enviar los comandos apropiados y ser capaz de interpretar los mensajes de datos trasmitidos por esta. En el capítulo 2 se puede obtener más información sobre la interfaz de comunicación con la FPGA.
- El software debe ser capaz de calcular el valor global de la estación y las correcciones que se realizan sobre este. Estos valores son explicados en la última sección de este capítulo.
- El software debe ser capaz de almacenar en local tanto los eventos por minuto de los 18 canales como el dato global y las correcciones de este.
- El software también debe ser capaz de mantener una réplica remota de los datos.
- Al igual que el hardware, el software debe ser capaz de adaptarse fácilmente a diferentes estaciones. Para este propósito este debe ser diseñado de tal forma que sea fácil de extender.
- El software debe ser capaz de poner en marcha la estación completa de forma automática ante la presencia de corriente eléctrica.
- El software debe ser capaz de detectar estados anómalos y actuar conforme a estos. En una estación de este tipo la mayoría de veces un funcionamiento anómalo se traduce en no generar datos o generar datos irregulares. Ante la presencia de un estado anómalo debe generarse una alerta. El software también debe reaccionar ante dichos estados anómalos, la mayoría de problemas se solucionan realizando un reinicio del sistema.

Aparte de la realización del software en este trabajo también contemplamos el proceso de implantación y mantenimiento de este. Volvemos a recordar que los demás módulos que componen el sistema de adquisición ya están desarrollados por el equipo de CaLMa. Las interfaces de estos son descritas, dado que es necesario para entender este trabajo.

1.5.2. Herramienta Web

El segundo objetivo de este trabajo es el desarrollo de una herramienta que facilite la gestión de una estación. La idea de esta es del equipo de CaLMa. Procedemos a detallar las funcionalidades de la herramienta tal y como las concibe este.

- **Spike Tool**. Módulo que permitirá la detección de Spikes. Un Spike es un dato anómalo, un dato anormalmente grande o pequeño. Usando los datos proporcionados por el sistema de adquisición, este módulo generará gráficos. Éstos serán interactivos y su propósito será hacer más fácil la detección de Spikes. Los Spikes detectados podrán ser marcados como nulos en el conjunto de datos revisados.

- **Configuración de los parámetros la estación**. Este módulo permitirá la reconfiguración de los parámetros de la estación. Esta reconfiguración será llevada a cabo sin interrumpir el proceso de adquisición. Nótese que el software de adquisición deberá evolucionar para hacer posible esta funcionalidad.

- **Alertas**. La herramienta visualizará las alertas producidas por el software de adquisición.

- **Histogramas con la intensidad de los eventos**. El nuevo sistema de adquisición proporciona información sobre la energía de las partículas detectadas. Este módulo generará histogramas con estos datos. Estos histogramas permitirán hacer mejores diagnósticos sobre el funcionamiento de los tubos contadores.

Podemos ver que la herramienta ofrece un gran abanico de funcionalidades. Por desgracia en este trabajo tan solo nos centraremos en el primer módulo, realizar los demás módulos está fuera del alcance de un trabajo como este. También cabe destacar que tan solo nos centraremos en la implementación, no implantaremos ni mantendremos la herramienta. La herramienta será una herramienta Web intuitiva y altamente interactiva.

1.6. Diseño preliminar

En esta sección procedemos a especificar un diseño preliminar para el software de adquisición y para la herramienta Web. La primera subsección es dedicada al software de adquisición, mientras que la segunda es reservada para la herramienta Web.

1.6.1. Software de adquisición

El sistema de adquisición que se está desarrollando es un sistema empotrado donde hardware y software están muy vinculados. Esto en gran medida condiciona el diseño del software que queremos realizar. El software debe ejecutarse sobre una *BeagleBone Black*[6] que está integrada con el resto de componentes hardware. La distribución Linux elegida para este trabajo es *Angstrom*, la distribución por defecto. El lenguaje de programación elegido es *Python*[19], lenguaje interpretado de alto nivel con tipado dinámico y sintaxis centrada en producir código legible. Actualmente *Python* es muy popular y existen muchas bibliotecas de las que haremos uso. El uso de bibliotecas reduce la carga de trabajo y normalmente resulta en software más robusto. Para la gestión de la base de datos hemos elegido *Sqlite3*[9], una elección popular en sistemas

empotrados. Este es un sistema ligero, de alto rendimineto que implementa la base de datos en un único archivo. Esto junto al API bien definido simplifica la administración.

En la figura 1.2 podemos ver el diseño preliminar del software de adquisición. Ante la presencia de corriente eléctrica este debe ser capaz de inicializarse automáticamente. El primer paso que debe hacer es realizar la configuración necesaria para el correcto funcionamiento del sistema de adquisición. Seguidamente debe continuar con el funcionamiento nominal. Este consiste de tres pasos que se repiten cíclicamente. El primero es solicitar la información a la FPGA, seguidamente el software debe interpretar dicha información y finalmente la información debe ser guardada. Los posibles estados anómalos deben ser detectados y contrarrestados.

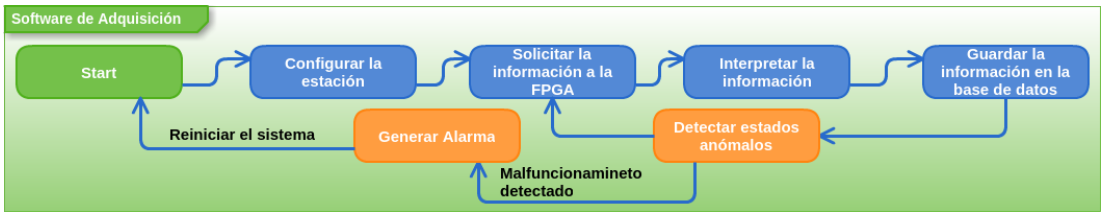

Figura 1.2: Software de adquisición. Diseño preliminar.

1.6.2. Herramienta Web

En la figura 1.3 podemos ver el diseño preliminar de la herramienta Web. Como podemos observar el diseño está fuertemente basado en el modelo *Cliente-Servidor*[10]. En este trabajo utilizamos los términos *back-end* y *front-end* para referirnos al *Cliente* y al *Servidor* respectivamente. A continuación explicamos los componentes básicos.

Base de Datos. En este componente residen los datos de nuestra estación. Para la gestión de estos utilizamos un servidor *MySQL*[1].

Back-End. Este componente es el encargado de recibir y procesar los mensajes de petición provenientes del *Front-End*. Estas peticiones pueden ser de consulta o de acción. En ambos casos este componente procede a comunicarse con la base de datos a fin de satisfacer la petición. Finalmente el resultado es trasmitido al *Front-End* en un mensaje de respuesta. En el caso de una petición de consulta son devueltos los datos especificados. En el caso de una petición de acción es devuelto un mensaje de estado. Para la implementación de este componente hemos utilizado *ZendFramework*[31] y *Apygility*[28].

Front-end. Este componente implementa la interfaz de nuestra aplicación. Es el encargado de presentar la información y manejar las peticiones del usuario. El módulo está basado en el patrón *Modelo-Vista-Controlador*. Para implementar la *Vista* encargada de presentar los datos hemos utilizado *HighStock*[3], para el *Controlador* responsable de gestionar los eventos empleamos *ExtJs*[22] y para el modelo encargado de manejar los datos utilizamos peticiones *Ajax*[38].

1.7. Proceso de adquisición

El propósito de este punto es explicar algunos aspectos del proceso de adquisición que son relevantes para este trabajo.

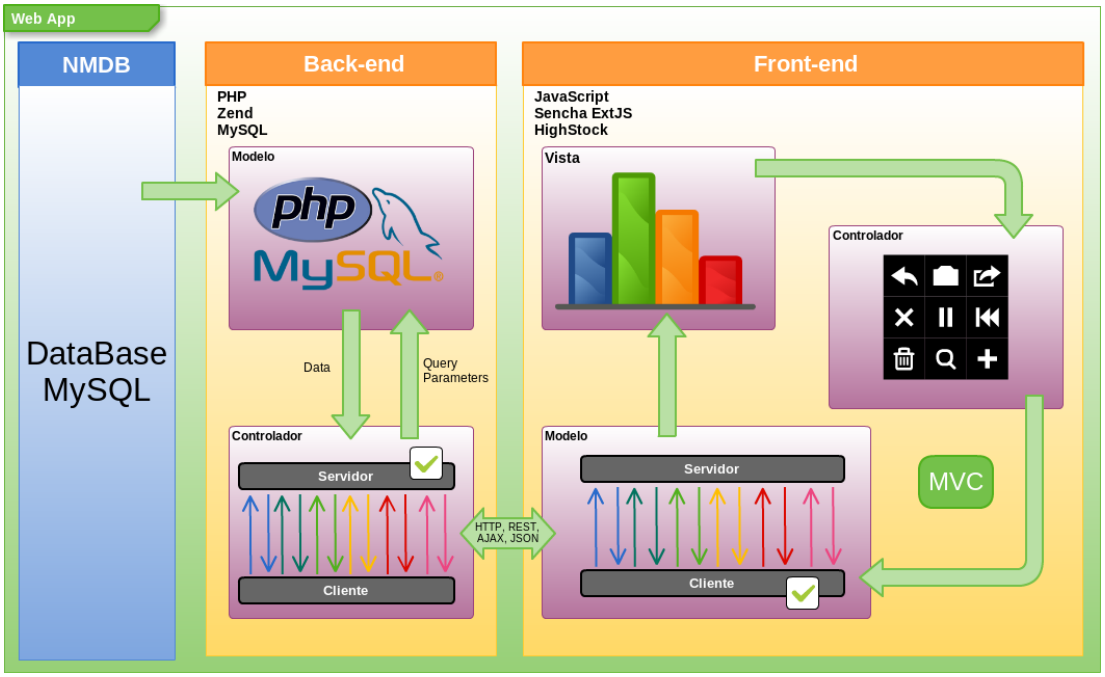

Figura 1.3: Herramienta Web. Diseño preliminar

Hasta este momento siempre hemos hablado de tubos contadores, en plural, detrás de esto hay una razón. Normalmente las estaciones se componen de varios tubos contadores, donde 18 tubos contadores es un estándar. En el proceso de adquisición están envueltos muchos factores probabilísticos, esto conlleva a que las medidas en un tubo tengan una gran dispersión. La solución de este problema es tener muchos tubos, cuantos más mejor. Combinando datos de diferentes tubos conseguimos reducir esta dispersión.

Tener los datos de muchos tubos contadores permite reducir la dispersión de los datos, sin embargo no es muy práctico trabajar con esos datos en crudo. El software del sistema de adquisición actual calcula un valor global a partir de los datos de todos los tubos contadores. El software de adquisición que desarrollamos en este trabajo también debe calcular este valor global. Para este propósito utilizamos el Median Algorithm[20].

Anteriormente en este capítulo explicamos las cascadas atmosféricas que son originadas por los rayos cósmicos. Estas cascadas atmosféricas dependen de la presión atmosférica. Cuando esta es elevada, es necesaria más energía para que la cascada se propague hasta el nivel terrestre, por consecuente los monitores de neutrones registran menos eventos. El valor de la presión atmosférica es monitorizado por los monitores de neutrones. Este valor es utilizado para realizar una corrección por presión sobre el valor global de la estación, a lo largo de este trabajo utilizamos el término de *valor corregido por presión* para referirnos a este valor. El software de adquisición elaborado para este trabajo debe ser capaz de leer el valor de presión desde un barómetro y debe poder calcular esta corrección.

También es realizada una *corrección por eficiencia*. Esta es muy simple y consiste en aplicar un factor multiplicativo al *valor corregido por presión*. Este valor es utilizado para para solventar problemas técnicos. Cambios en el entorno inmediato del instrumento o cambios en la electrónica utilizada pueden afectan a la cantidad de eventos medidos. Estos cambios son identificados, evaluados y finalmente contrarrestados con esta

corrección por eficiencia. Este valor dota la estación de consistencia histórica, de esta manera pueden ser comparados datos de diferentes intervalos temporales. El software también debe ser capaz de realizar esta corrección.

Al principio de este capítulo explicamos que los tubos contadores están rellenos de gas con propiedades especiales. Sobre este gas es aplicado un campo eléctrico. Para la generación de dicho campo son utilizadas fuentes de alimentación de alta tensión. Cambios en la tensión generada pueden afectar a la cantidad de eventos registrados. Esto ha conllevado a que el funcionamiento de las fuentes sea monitorizado. El software para el sistema de adquisición debe ser capaz de realizar esta operación. Es de esperar que la tensión sea constante, en caso de variaciones los datos generados son considerados no consistentes.

Un *Spike* es un dato anómalo, un dato anormalmente grande o pequeño. Son datos producidos por mal funcionamientos de la instrumentación, cambios bruscos en el entorno inmediato u otros factores desconocidos. Estos están presentes en todas las estaciones. Con la elaboración del nuevo sistema de adquisición se pretende reducir el número de *Spikes* generados. Con la elaboración de la herramienta Web pretendemos ofrecer un método fácil de identificar y descartar dichos datos.

Capítulo 2

Entorno Hardware

En un sistema empotrado el software y hardware están muy vinculados. Para entender el funcionamiento del software de adquisición es muy importante estar familiarizado con el diseño y funcionalidad del hardware. Impulsados por esta razón en este capítulo procederemos a hacer una descripción de los aspectos más importantes del hardware que compone el sistema de adquisición. Destacamos que el desarrollo de estos módulos está fuera del ámbito de este proyecto.

2.1. BeagleBone Black

BeagleBone Black[6][8] es un computador empotrado, *open-source* y *single board*. La placa viene con Linux, distribución *Angstrom* y versión de núcleo 3.8. En la figura 2.1 podemos ver un diagrama de bloques que refleja los componentes hardware que componen la *BeagleBone Black*. A continuación vamos a hacer una breve descripción de los módulos que utilizamos para este trabajo.

- ARM CORTEX A8[25] y SDRAM. El procesador es suficientemente potente para soportar la distribución Linux y satisfacer las necesidades de nuestro software. Los 512MB DDR3 de memoria RAM son suficientes para los propósitos de este trabajo.

- Ethernet Connector. El conector RJ-45 permite establecer una conexión a Internet. Una vez establecida esta conexión los datos pueden ser transmitidos al NMDB. También es posible establecer una conexión a la placa vía SSH, de esta manera un operador puede realizar operaciones de mantenimiento de forma remota.

- eMMC y microSD. Utilizamos la memoria integrada (eMMC) de 4GB para albergar el sistema operativo y el software de adquisición. La microSD es utilizada para guardar los datos y logs producidos por el software de adquisición.

- Analog Pins. Los 7 pines analógicos permiten usar sensores cuyo output es una señal analógica. Ejemplo de estos sensores son las fuentes de alimentación analógicas o los sensores de temperatura, en algunas estaciones la temperatura ambiente se monitoriza también. Estos sensores producen señales cuya tensión eléctrica es proporcional al valor de la magnitud medida.

- GPIOs. Los *General Purpose Input/Output* permiten trabajar con señales digitales, tanto de entrada como de salida. Un ejemplo de uso es la señal de Reset de la FPGA, señal digital activa a bajo nivel.

- UARTs. La placa ofrece 4 puertos serie de los que utilizamos 2 para comunicarnos con la FPGA.

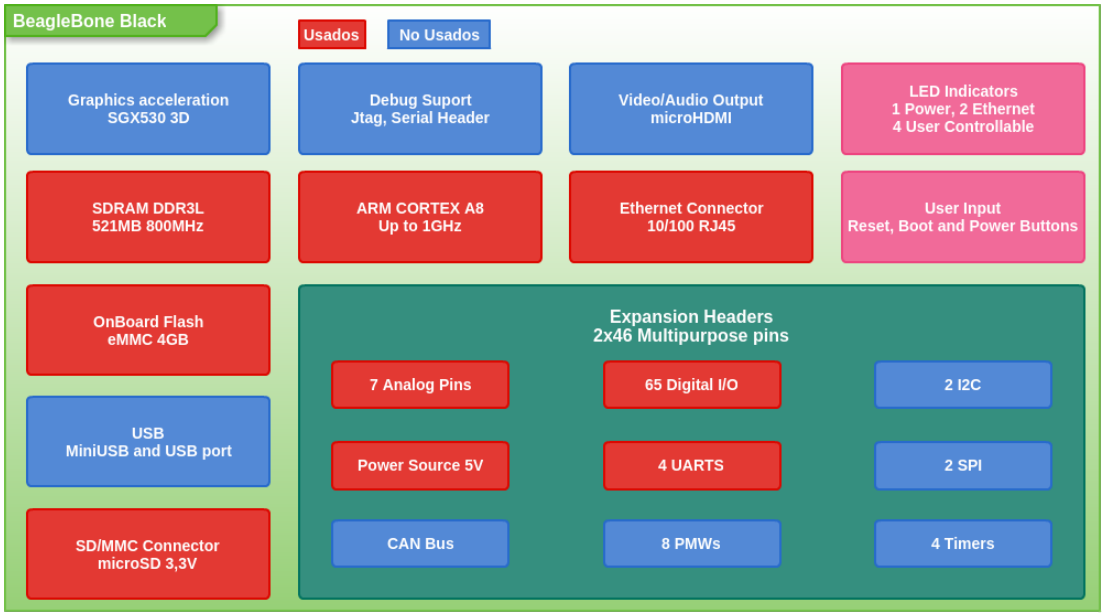

Figura 2.1: *BeagleBone Black*. Diagrama de bloques hardware

Fijándonos en la figura 2.1 podemos ver que muchos de los módulos son accesibles mediante los conectores de expansión[7]. Estos conectores tienen 2x46 pines, de los cuales muchos son multipropósito. Estos pueden ser configurados para tener funcionalidades diversas y además esta configuración puede ser realizada de forma dinámica. La configuración de los pines se realiza mediante el *Device Tree Overlay*, estructura de datos que se utiliza para describir el hardware. En la realización de este trabajo no lidiamos con este mecanismo, sino que utilizamos una biblioteca que facilita el trabajo. La biblioteca utilizada es *Adafruit BeagleBone IO Python*[26]. Esta ayuda a configurar dinámicamente los pines de expansión, además ofrece métodos para operar sobre estos una vez configurados.

2.2. FPGA

La FPGA es el núcleo fundamental del sistema, ya que se encarga de procesar las señales y de transmitir la información al software de adquisición. Una FPGA es un circuito integrado formado por unidades lógicas cuya interconexión y funcionalidad pueden ser configurados mediante un lenguaje de descripción de hardware, típicamente *Verilog* o *VHDL*. Esta configuración del hardware de la FPGA, también conocida como *IP core*, es la que establece la funcionalidad de una FPGA. En esta sección vamos a hacer una descripción funcional de la FPGA, y por consiguiente se describirá el *IP core* de la misma. En la figura 2.2 podemos ver un diagrama de bloques que refleja los módulos e interfaces que el *IP core* posee.

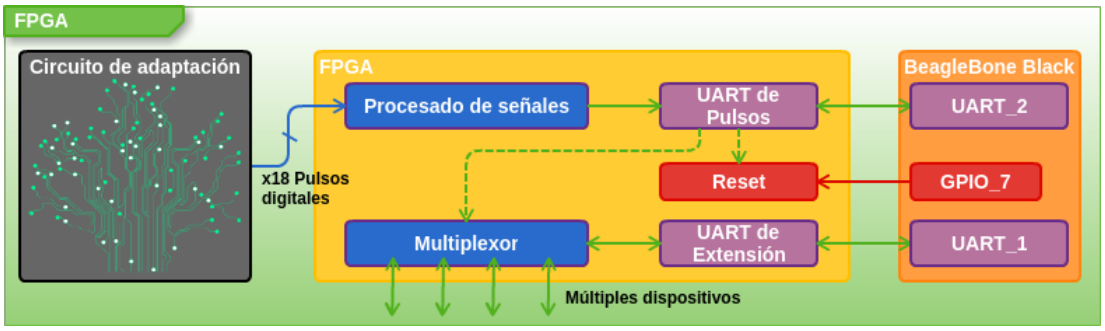

Figura 2.2: FPGA. Diagrama de bloques.

2.2.1. Procesado de pulsos

Como hemos comentado la FPGA es la encargada de procesar las señales y trasmitir la información a la *BeagleBone Black*. Esta transmisión se realiza mediante una línea serie asíncrona de 1Mbps. La línea está controlada por una UART(*Universal Asynchronous Receiver-Transmitter*) a la que nos referiremos como *UART de pulsos*. En las tablas 2.1, 2.2 y 2.3 podemos ver el formato de los datos transmitidos. Vemos que hay tres mensajes diferentes que la FPGA puede transmitir.

- En la tabla 2.1 podemos ver el formato del primer mensaje que vamos a discutir. Este mensaje se compone de tres bytes que dan información sobre el ancho de un pulso. El mensaje permite identificar el canal en el que se ha producido el pulso, el nivel (alto/bajo) y la longitud de este. Los pulsos de nivel alto representan la detección de una partícula por los tubos contadores y el ancho del pulso es proporcional a la energía de la partícula detectada. La longitud de los pulsos de nivel bajo también se mide para poder identificar pulsos con pequeña separación temporal, que podrían indicar la presencia de un fenómeno físico denominado multiplicidad.

- En la siguiente tabla 2.2 podemos apreciar el mensaje de estado que la FPGA genera. Dado que la UART usa una FIFO de tamaño fijo es posible que la esta se llene y se pierdan mensajes. En este caso se genera un mensaje de estado que es transmitido para indicarle al software que se han producido perdidas de datos.

- Además de medir el ancho de los pulsos la FPGA lleva la cuenta de pulsos recibidos para cada canal. Esta información puede ser solicitada por la *BeagleBone Black* utilizando el comando apropiado(ver tabla 2.4). En la tabla 2.3 podemos ver el formato en el que se transmite la información de las cuentas. Los bytes 2, 3 y 4 son transmitidos 18 veces, una vez para cada canal. Después de transmitir la información de las cuentas la FPGA reinicia los contadores para cada canal.

En la figura 2.3 podemos observar un ejemplo de la secuencia de datos transmitidos por la *UART de pulsos*, junto a estos podemos apreciar los estímulos que los causan.

2.2.2. Multiplexor

Tal y como explicamos en el capítulo de introducción, el sistema de adquisición debe monitorizar la presión atmosférica, el funcionamiento de las fuentes de tensión y en muchos casos la temperatura ambiente. Al igual que muchos otros dispositivos que pueden ser necesarios, la mayoría de estos sensores hacen uso de un puerto serie. Estos

Bit	7	6	5	4	3	2	1	0
Byte 1	0	1	0	Canal (0-17)				
Byte 2	1	Dato (6)	Dato (5)	Dato (4)	Dato (3)	Dato (2)	Dato (1)	Dato (0)
Byte 3	1	X	Nivel ('1'->alto, '0'->bajo)	Dato (11)	Dato (10)	Dato (9)	Dato (8)	Dato (7)

Cuadro 2.1: *UART de pulsos*. Palabra de ancho de pulso

Bit	7	6	5	4	3	2	1	0
Byte 1	0	0	OverFlow FIFO Tubo 5	OverFlow FIFO Tubo 4	OverFlow FIFO Tubo 3	OverFlow FIFO Tubo 2	OverFlow FIFO Tubo 1	OverFlow FIFO Tubo 0
Byte 2	1	OverFlow FIFO Tubo 12	OverFlow FIFO Tubo 11	OverFlow FIFO Tubo 10	OverFlow FIFO Tubo 9	OverFlow FIFO Tubo 8	OverFlow FIFO Tubo 7	OverFlow FIFO Tubo 6
Byte 3	1	Almost Full FIFO General	OverFlow FIFO General	OverFlow FIFO Tubo 17	OverFlow FIFO Tubo 16	OverFlow FIFO Tubo 15	OverFlow FIFO Tubo 14	OverFlow FIFO Tubo 13

Cuadro 2.2: *UART de pulsos*. Palabra de estado

Bit	7	6	5	4	3	2	1	0
Byte 1	0	1	1	X	X	X	X	X
Byte 2	1	Dato0 (6)	Dato0 (5)	Dato0 (4)	Dato0 (3)	Dato0 (2)	Dato0 (1)	Dato0 (0)
Byte 3	1	Dato0 (13)	Dato0 (12)	Dato0 (11)	Dato0 (10)	Dato0 (9)	Dato0 (8)	Dato0 (7)
Byte 4	1	X	X	X	X	X	Dato0 (15)	Dato0 (14)
Byte 5	1	Dato1 (6)	Dato1 (5)	Dato1 (4)	Dato1 (3)	Dato1 (2)	Dato1 (1)	Dato1 (0)
.....	.	.....	.....	.....	.....	.....	.....	.....
Byte 55	1	X	X	X	X	X	Dato17 (15)	Dato17 (14)

Cuadro 2.3: *UART de pulsos*. Palabra de cuentas

Commando	Descripción
0x00	Configura multiplexor para aparato 1
0x01	Configura multiplexor para aparato 2
0x02	Configura multiplexor para aparato 3
0x03	Configura multiplexor para aparato 4
0x10	Reset general del sistema
0x11	Solicita la transmisión de las cuentas

Cuadro 2.4: *UART de pulsos*. Commandos

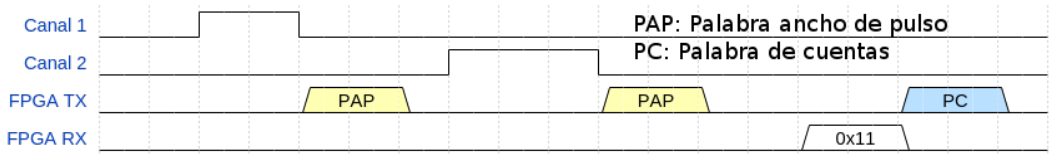

Figura 2.3: FPGA. Diagrama de eventos.

son manejados según el patron *Master-Slave* y generalmente intercambian poca información, por consiguiente, pueden compartir un puerto serie. El mecanismo que resuelve este problema está implmentado en el *IP core*. Este consiste en una UART a la que nos referiremos como *UART de extensión*. Esta está conectada a un multiplexor, que es el encargado de realizar la conmutación entre los cuatro dispositivos soportados. El estado del multiplexor puede cambiarse enviando comandos por la *UART de pulsos*, de acuerdo a la especificación presentada en la tabla 2.4. Ejemplo de dispositivos que requiren un puerto serie son el barómetro *BM35* utilizado en CaLMa, varias fuentes de alimentación comerciales, estaciones meteorológicas, GPS, etc.

2.2.3. Reset

El *IP core* permite realizar un *Reset* del estado interno de la FPGA. Todas las variables son puestas a sus valores iniciales, excepto el multiplexor que mantiene su estado. Para la realización de dicho Reset son exportadas tres interfaces. La primera es un boton hardware accesible de forma física. La segunda es un commando trasmitido por la *UART de pulsos*, de acuerdo a la especificación presentada en la tabla 2.4. Finalmente, la tercera es una señal digital activa a bajo nivel. Dicha señal definida como *BS1*, está conectada al pin *P9_42(GPIO_7)* de la *BeagleBone Black*, por consiguiente, es accesible desde nuestro software de adquisición.

Capítulo 3

Software de adquisición

En este capítulo procedemos a explicar el funcionamiento del software de adquisición. Empezaremos recordando los aspectos básicos descritos en el capítulo de introducción. El funcionamiento nominal del software consiste en recoger la información de los módulos hardware como la FPGA, el barómetro, las fuentes de alimentación y los sensores de temperatura si estos están presentes. Cada minuto, la información recogida es procesada y almacenada en una base de datos.

Antes de empezar a discutir a fondo el software de adquisición es conveniente estar familiarizados con el entorno hardware descrito en el capítulo 2. El software es ejecutado en la *BeagleBone Black* sobre un Linux, distribución *Angstrom*. El lenguaje elegido para el desarrollo es *Python*[19]. Este es un lenguaje interpretado de alto nivel con una sintaxis centrada en producir un código legible. El lenguaje se adapta igualmente bien a programación imperativa y orientada a objetos. El tipado dinámico da mucha flexibilidad al lenguaje. Todas estas propiedades del lenguaje permiten un desarrollo rápido. Además el lenguaje ofrece una amplia cantidad de bibliotecas que también aligeran el trabajo.

Para gestionar la base de datos utilizamos *Sqlite3*[9]. Este es un gestor muy ligero y adecuado para sistemas empotrados. Aparte de esta base de datos local, el software permite mantener una segunda réplica remota. Para esta réplica remota utilizamos *MySql*[1] que es un gestor más completo con enfoque *Cliente-Servidor*.

En la figura 3.1 podemos ver un diagrama de flujo que describe el funcionamiento de nuestro software. A lo largo de este capítulo haremos muchas referencias a este diagrama para explicar los diferentes módulos software que lo componen. Podemos ver que hay cuatro módulos principales, a continuación hacemos una breve descripción de estos.

- `NMDA`. Es el proceso principal de nuestro software. Es el encargado de realizar las configuraciones necesarias e iniciar el `FPGASerialReader` y el `CountsManager`.
- `FPGASerialReader`. *Thread* que se encarga de procesar la información transmitida por la FPGA.
- `CountsManager`. *Thread* que se enarga de pedir la información necesaria y guardarla en la base de datos con periodicidad de un minuto. Después de guardar la información es iniciado el `DBUpdater`.
- `DBUpdater`. *Thread* que se encarga de sincronizar la base de datos local con la réplica remota.

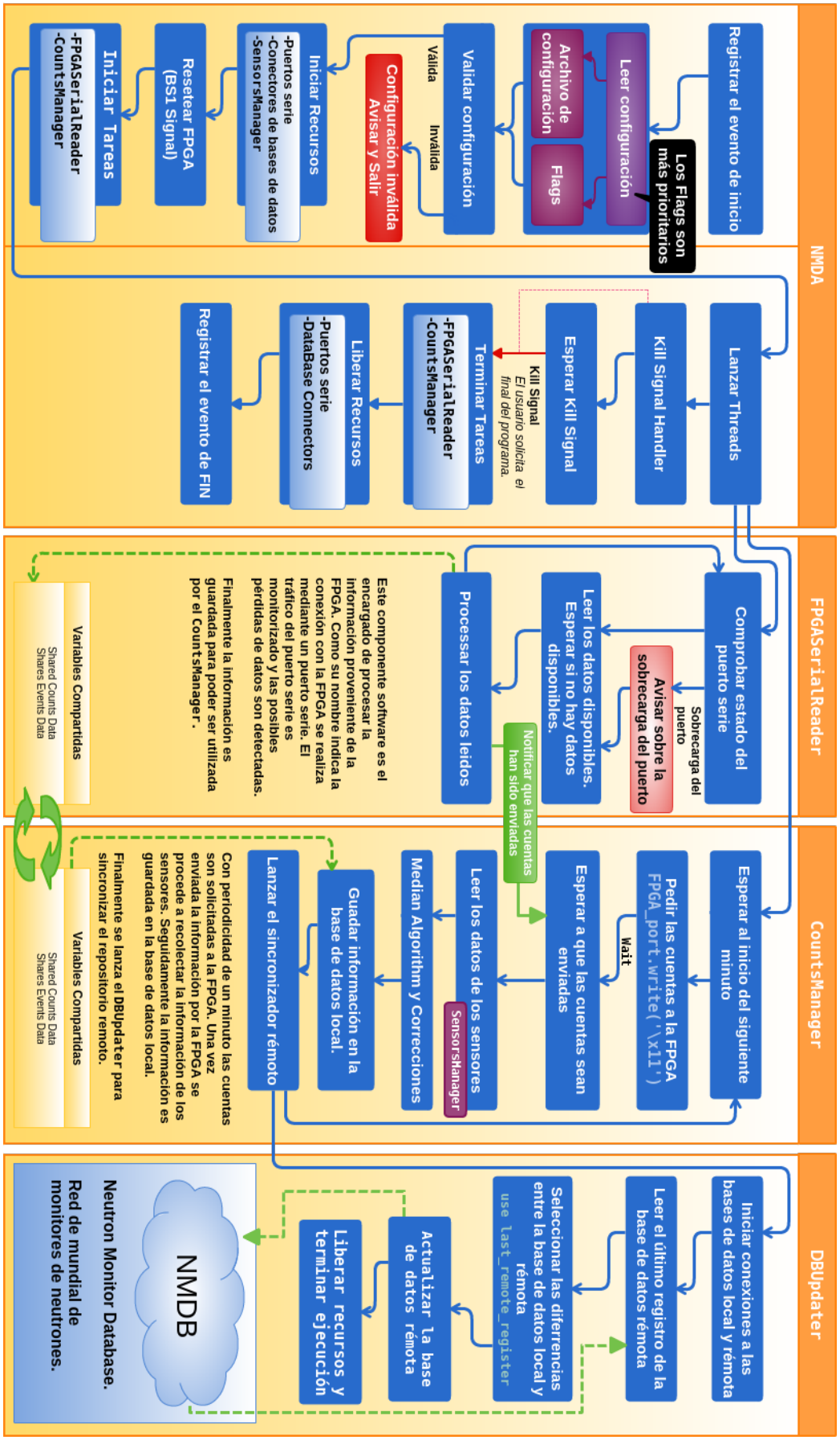

Figura 3.1: Diagrama de flujo. Software de adquisición.

3.1. Arranque automático del sistema

Uno de los requisitos del sistema es el arranque automático ante la presencia de corriente eléctrica. La *BeagleBone Black* por defecto está configurada para arrancar automáticamente,pero debemos configurarla para que también inicialice nuestro software. Para este propósito vamos a utilizar las *System Services*[37] que *Angstrom* proporciona. Esta distribución utiliza `systemd`[49] para gestionar los servicios de sistema y la forma en la que estos servicios arrancan cuando se pone en marcha la *BeagleBone Black*. Este es un conjunto de demonios, bibliotecas y utilidades diseñados para facilitar la administración de sistemas Linux.

Para operar sobre la configuración de `systemd` tenemos el comando `systemctl`. A continuación podemos ver como usar este comando.

```
$ systemctl start    nombre_de_servicio
$ systemctl restart  nombre_de_servicio
$ systemctl stop     nombre_de_servicio
$ systemctl status   nombre_de_servicio
$ systemctl enable   nombre_de_servicio
$ systemctl disable  nombre_de_servicio
```

Los servicios de sistema se crean mediante archivos con extensión `.service` que deben ser guardados en el directorio `/lib/systemd/system`. En el apéndice A.3 podemos ver el contenido de los archivos que definen los cuatro servicios de sistema utilizados en este trabajo. Veamos el significado de lo campos más importantes que son definidos en estos archivos.

- `Description`. Un mensaje descriptivo del servicio. Permite al usuario entender el propósito de este de forma fácil.
- `Before` y `After`. Estos dos campos pueden referenciar a otros servicios para indicar que este debe ejecutarse antes o después del servicio que es citado. Esto permite definir un orden en el que serán ejecutados los servicios de sistema. También es posible especificar un grupo de servicios como el `network.target` que agrupa los servicios responsables de establecer la conexión a Internet.
- `Type`. El tipo del servicio. Con este campo podemos dar diferentes propiedades a nuestro servicio. Los valores aceptados que utilizamos son `oneshot`, `forking` y `simple`, que es el valor por defecto.
- `ExecStart`. El comando y argumentos que queremos ejecutar.
- `WantedBy`. Es el grupo al que este servicio pertenece. El grupo `multi-user.target` es un grupo para los servicios de un sistema multiusuario. Este campo hace que este servicio sea retrasado hacia el final del proceso de arranque, específicamente hasta que el sistema sea listo para aceptar el acceso de múltiples usuarios.

En la figura 3.2 podemos ver la secuencia que define el proceso de arranque del sistema de adquisición. Esta secuencia empieza con el servicio `myWatchDog.service`, que es el encargado de arrancar el software que controla el *Watchdog*. El servicio `ntpdate.service` es el responsable de establecer la fecha y hora actuales mientras que el servicio `ntpd.service` tiene como tarea corregir las derivas que pueden presentrase. Finalmente el servicio `nmda.service` es el encargado de arrancar el software de adquisición.

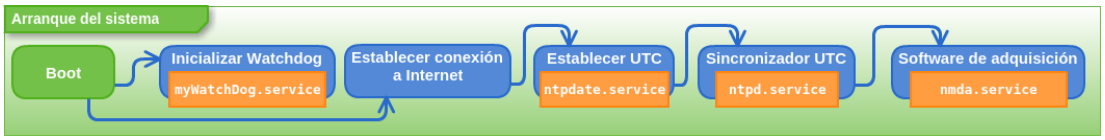

Figura 3.2: Arranque del sistema

3.1.1. Watchdog

Un *Watchdog* [52] es un mecanismo de seguridad que realiza un *Reset* en el sistema cuando es detectado un mal funcionamiento. Consiste en un temporizador que realiza una cuenta atrás, de forma que, cuando esta cuenta llega a cero, el sistema es reiniciado. Es nuestro software el que debe reponer el valor de este contador para evitar que expire. Los *Watchdogs* pueden ser software o hardware, el utilizado en este trabajo está implementado en la *BeagleBone Black* y es hardware. Los *Watchdogs* hardware son más fiables, los software son más susceptibles a bloquearse.

El *Watchdog* es accesible mediante el archivo `/dev/watchdog`. Para activar el mecanismo basta con escribir cualquier valor en dicho archivo. El contador se pone en marcha automáticamente y, si este llegara a expirar, se produciría el reinicio del sistema. El valor inicial del contador es de 60 segundos, valor que adquiere cada vez que es reiniciado. El contador es reiniciado al volver a escribir en el archivo. Al cerrar dicho archivo el mecanismo es deshabilitado. A continuación presentamos una parte del código fuente responsable de controlar el *WatchDog*.

```
1  args             = NMDA.create_parser().parse_args()
2  conn_local       = sqlite3.connect(args.database)
3  last_data        = get_last()
4  cont             = 0
5
6  wd = open("/dev/watchdog", "w+")
7  while True:
8      time.sleep(20)
9      wd.write("\n")
10     wd.flush()
11     cont+=1
12
13     if cont >= 15:              #15*20secs=5mins
14         cont             = 0
15         curr_last        = get_last()
16         if curr_last == last_data:
17             time.sleep(120)    #Este sleep reinicia la placa
18         last_data        = curr_last
```

Primero se establece la conexión con la base de datos local. La base de datos es utilizada para comprobar si el sistema de adquisición genera correctamente datos. Si no fuera así, la placa debe ser reiniciada. Una vez establecida la conexión se procede a invocar la función `get_last()`, que devuelve el último valor presente en la base de datos. Seguidamente se abre el archivo que permite manejar el *WatchDog*. Una vez abierto el archivo se entra en un bucle `while` sin condición de salida. En cada iteración de este bucle se escribe un salto de línea que reinicia el contador del *WatchDog*. Dentro del bucle también hay una llamada a la función `time.sleep(20)`, las iteraciones del bucle están espaciadas 20 segundos. Cada 15 iteraciones o aproximadamente 5 minutos se vuelve a invocar la función `get_last()`. El valor devuelto es comparado con el valor

anterior. Si estos dos valores son iguales es invocada la función `time.sleep(120)`, intervalo suficiente para que el contador del *WatchDog* expire. Los valores son iguales si en los últimos 5 minutos el sistema de adquisición no ha generado ningún dato. Por contrario si se han generado datos durante los últimos 5 minutos estos dos valores son diferentes, en tal caso se sigue con la ejecución del bucle `while`.

3.1.2. Sincronización de fecha y hoara

La *BeagleBone Black* no implementa ningún mecanismo hardware para mantener la fecha y hora actuales, por lo que en cada reinicio la fecha y hora se pierden. Es nuestra responsabilidad realizar un mecanismo software que sincronice la fecha y hora con el resto del mundo[35]. Utilizamos el programa `ntpd` para este propósito. Este programa hace uso del *Network Time Protocol*[47], que permite la sincronización entre dos relojes a través de una red de datos con latencia variable. Son realizadas dos llamadas a este programa.

```
# Establece inicialmente la fecha y hora.
$ /usr/bin/ntpd -q -g -x
# Lanza un demonio que corrige las derivas que puedan presentarse
$ /usr/bin/ntpd -p /run/ntpd.pid
```

3.2. NMDA

Este es el módulo principal de nuestro software. Es el encargado de realizar la configuración inicial de acuerdo con los parámetros proporcionados, resetear la FPGA y finalmente inicializar los demás módulos. En la figura 3.1 podemos ver un diagrama de flujo que representa el funcionamiento de este. En las siguientes subsecciones describiremos las funcionalidades más relevantes de este módulo.

3.2.1. Logger

Haciendo uso de la biblioteca `logging`[16] se configura un archivo de Log. En este archivo se registran los sucesos de eventos importantes. Un ejemplo de los posibles eventos que se registran son los relacionados con las pérdidas de datos. El mecanismo está configurado para que automáticamente se guarde la hora y fecha de cada mensaje. A continuación se muestra cómo se configura y usa este mecanismo. Los comentarios representan la entrada producida en el archivo de Log.

```
1 import logging
2 logging.basicConfig(filename='/server/logs/NMDA.log', level=
     logging.DEBUG, format=''%(asctime)s %(message)s'')
3 logging.info('Data Loss')
4 # 2015-05-14 09:50:56,017   Data Loss
5 logging.info('Uart_1 Overflow')
6 # 2015-05-14 09:50:55,936   Uart_1 Overflow
```

3.2.2. Archivo de configuración y Flags

Existen una serie de parámetros que varían en función de la estación o que simplemente pueden cambiar con el tiempo. No es buena idea tener los valores de estas variables en el código. Este problema nos ha llevado a exportar estos

parámetros a un archivo de configuración. Para leer este archivo utilizamos la biblioteca `ConfigParser`[15]. A continuación mostramos un ejemplo de este archivo que nos ayudará a entender mejor su estructura. Los comentarios explicativos son suficientes para entender el propósito de cado uno de los campos.

```
[Basics]
# Referencia a la Uart de pulsos dentro del sistema de archivos.
serial_port_control  = /dev/ttyO2
# Referencia la la base de datos local.
# El valor 'shell' imprimirá los valores por consola.
database = /server/data/test.db
# Valores medios para los 18 canales que serán utilizados para el
    Median Algorithm.
Channel_avg = 255, 290, 0, 295, 0, 0, 289, 291, 252, 254, 293,
   299, 298, 328, 299, 302, 302, 272

[Sensors]
# Referencia a la Uart de extensión dentro del sistema de
   archivos.
serial_port_sensors= None
# Tipo de barómetro. Valores aceptados [None, bm35, ap1].
barometer_type = None
# Tipo de HVPS. Valores acpetados [None, analog, digital].
hvps_type = None
# Coeficiente de correción para las HVPS analógicas.
analog_hvps_corr = 1.0

[dbUpdater]
# Habilitar y deshabilitar.
db_updater_enabled=True
# Referencia la la base de datos local. Debe tener el mismo valor
    que Basics.database.
local_db= /server/data/test.db
# La dirreción IP de la base de datos remota.
remote_db_host= 192.168.1.1
# Usuario y contraseña del usuario para la base de datos remota
remote_db_user= hristo
remote_db_pass= 123qwe
# Nombre de la base de datos remota
remote_db_db= nmdadb2

[Pressure]
# Presión media para la estación
avg_pressure=932
# Coeficiente de correlación atmosférica
beta_pressure=0.0067

[Efficiency]
# Factor de correción
beta_efficiency=1.0
```

Los valores de los parámetros exportados son también accesibles a través de flags. En este caso hemos utilizado la biblioteca `argparse`[14]. Los valores especificados mediante flags tienen precedencia sobre los mismos que estuvieran en el archivo de configuración. Los flags se usan de la siguiente forma.

```
$ python NMDA.py -h
usage: NMDA.py
   [-h] [-sp SERIAL_PORT_CONTROL] [-db DATABASE]
   [-sps SERIAL_PORT_SENSORS] [-bm {ap1,bm35}]
   [-hv {digital,analog}] [-ahvc ANALOG_HVPS_CORR]
   [-dbU DB_UPDATER_ENABLED] [-ldb LOCAL_DB] [-rh REMOTE_DB_HOST]
   [-ru REMOTE_DB_USER] [-rp REMOTE_DB_PASS] [-rdb REMOTE_DB_DB]
   [-apr AVG_PRESSURE]
```

3.2.3. Configurar la BeagleBone Black

Tal y como explicamos en el capítulo de entorno hardware la *BeagleBone Black* proporciona dos conectores de expansión de 46 pines cada uno[7]. Muchos de estos pines son multipropósito, es decir, que están compartidos por varias unidades funcionales internas, por lo que debe configurarse a cuál de ellas se conectará. Esta configuración se realiza mediante el uso de *Device Tree Overlay*, que es una estructura de datos utilizada para describir el hardware.

En este trabajo utilizamos la biblioteca *Adafruit BeagleBone IO Python*[26] que facilita la tarea de configurar estas cabeceras. A continuación podemos ver una lista de los pines utilizados y la función de cada uno en el ámbito de este proyecto.

```
Pin P9_21(UART2_TXD) --> Uart de Control trasmisión
Pin P9_22(UART2_RXD) --> Uart de Control recepción
Pin P9_24(UART1_TXD) --> Uart de Extensión trasmisión
Pin P9_26(UART1_RXD) --> Uart de Extensión recepción
Pin P9_37(AIN2)      --> Entrada analógica(sensores analógicos)
Pin P9_38(AIN3)      --> Entrada analógica(sensores analógicos)
Pin P9_39(AIN0)      --> Entrada analógica(sensores analógicos)
Pin P9_40(AIN1)      --> Entrada analógica(sensores analógicos)
Pin P9_42(GPIO_7)    --> Reset de la FPGA.
Pin P8_08(GPIO_67)   --> Señal DATA del barómetro AP1.
Pin P8_09(GPIO_69)   --> Señal CLOCK del barómetro AP1.
Pin P8_10(GPIO_68)   --> Señal STROBE del barómetro AP1.
```

3.2.4. Iniciar recursos

Al ser el módulo principal, `NMDA` es el encargado de iniciar todos los recursos que serán utilizados. Por recursos nos referimos a las variables compartidas entre *threads*, conectores a las bases de datos, interfaces de los puertos serie y otros. Para los conectores a las bases de datos utilizamos las bibliotecas `sqlite3` y `MySQLdb`, volvemos a recordar que tenemos una réplica local con *Sqlite3* y otra réplica remota que gestionamos con *MySql*. Las tablas son creadas automáticamente en caso de no existir, esto simplifica mucho el proceso de implantación. Para la interfaz de los puertos serie utilizamos la biblioteca `serial`.

Son inicializados los otros dos *threads*, `FPGASerialReader` y `CountsManager`. Estos dos realizan todo el proceso de adquisición y serán descritos más adelante en este capítulo.

3.2.5. Resetear FPGA

Como hemos explicado en el capítulo 2, la FPGA permite realizar un *Reset*, que el software realizará antes de empezar con la adquisición de datos. Este *Reset* asegura que la FPGA está en un correcto estado. Para realizar el *Reset* utilizamos la señal digital, *BS1*, que está conectada al pin *P9_42* de la *BeagleBone Black*. Volvemos a recordar que esta es activa a nivel bajo. Para controlarla utilizaremos la biblioteca de *AdaFruit*[26], que gestiona las entradas y salidas de propósito general(GPIO) de la siguiente forma.

```
1 import Adafruit_BBIO.GPIO as GPIO
2 GPIO.setup('P9_42', GPIO.OUT)
3 GPIO.output('P9_42', GPIO.LOW)
4 time.sleep(0.5)
5 GPIO.output('P9_42', GPIO.HIGH)
```

3.2.6. Kill Signal Handler

Este es un software que debe ejecutarse indefinidamente. En su uso real no será interrumpido, pero hemos implementado un mecanismo que permite pararlo. Este mecanismo permite solicitar el fin del programa, que desencadena una serie de acciones. Estas acciones tienen como objetivo asegurar que todos los recursos sean liberados de forma correcta. Liberar los recursos de forma correcta supone no tener ningún conflicto si seguidamente volvemos a ejecutar el software. Esto en el uso real del software no tiene gran impacto, sin embargo ha sido muy útil durante el proceso de desarrollo.

3.3. `FPGASerialReader`

Este módulo software es el encargado de leer y procesar los datos transmitidos por la FPGA. En la figura 3.1 podemos ver un diagrama de flujo que refleja el funcionamiento de este. La ejecución del *thread* empieza comprobando el estado del puerto serie, que puede estar saturándose. En el caso de que el puerto esté saturado se genera una entrada en el archivo de Log indicando este hecho. A continuación leemos los datos disponibles, y en caso de no haber datos disponibles el *thread* se queda bloqueado esperando a la llegada de estos. Seguidamente de leer los datos pasamos a procesarlos. Por procesar nos referimos a interpretar el significado que tienen. Finalmente volvemos al paso inicial, volviendo a comprobar si han llegado nuevos datos a la UART.

En las tablas 2.1, 2.2 y 2.3 del capítulo 2, podemos ver el formato de los datos que tenemos que procesar. Vemos que hay tres tipos de mensajes que podemos recibir. Los mensajes no siguen un orden concreto, además llegan de forma totalmente asíncrona. Además el canal de trasmisión no asegura la correcta trasmisión de los bytes por lo que algunos pueden perderse. Todo esto conlleva a que procesar los datos no sea una tarea fácil.

Para solucionar este problema hemos implementado una máquina de Moore. Si nos volvemos a fijar en el formato de los datos podemos ver que los primeros bits de cada byte son fijos. Estos bits ayudan a identificar a que mensaje corresponde el byte. Los estados y transiciones de la máquina pueden verse en la figura 3.3. El estado inicial es `ByteX`. Aunque no esté reflejado en la figura la recepción de cualquier valor no esperado nos lleva al estado inicial. También podemos ver que la lectura correcta de un mensaje de cuentas es notificada al `CountsManager`.

La información generada al procesar los datos es almacenada en las variables compartidas. Estas variables son accesibles desde el `CountsManager`, que se describe en la siguiente sección.

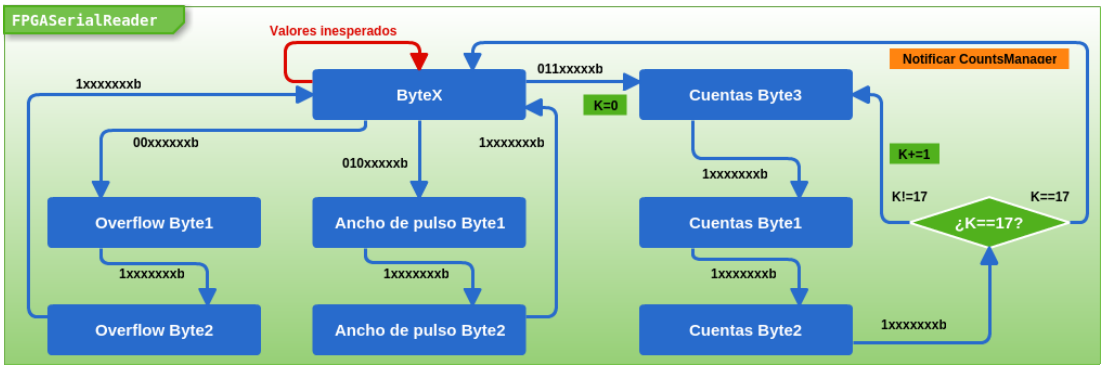

Figura 3.3: Máquina de Moore. `FPGASerialReader`.

3.4. CountsManager

Este módulo software se encarga de recolectar la información que el `FPGASerialReader` y `SensorsManager` generan. Seguidamente procede a calcular el valor global y las correcciones por presión y efficiencia. Finalmente se encarga de guardar los datos en la base de datos local. En la figura 3.1 podemos ver el diagrama que describe el funcionamineto que acabamos de explicar. A continuación procedemos a explicar a fondo estos pasos.

3.4.1. Recolectar información

Como vimos en el capítulo 2, la FPGA tan sólo transmite los datos de las cuentas cuando estas son solicitadas con el comando apropiado. Este módulo es el encargado de mandar el comando al principio de cada minuto. La información transmitida por la FPGA es procesada por el `FPGASerialReader` y almacenada en variables compartidas accesibles por el `CountsManager`. Después de mandar el comando apropiado para solicitar las cuentas este *thread* se queda esperando hasta que el `FPGASerialReader` le notifique que las cuentas han sido enviadas y procesadas. Al ser notificado este módulo procede a leer de las variables compartidas donde está almacenada la información de las cuentas. Dicha notificación se realiza mediante un `threading.Lock`

Los valores de presión atmosférica, diferencia de potencial generado por las fuentes de alimentación y temperatura ambiente son obtenidos haciendo uso del `SensorsManager`, que será descrito en secciones futuras de este capítulo.

3.4.2. Median Algorithm y Correcciones

Como hemos comentado al principio de esta sección este módulo calcula el valor global y una serie de correcciones sobre este. El valor global es una representación de las mediciones de todos los tubos, algo como una media. En el capítulo 1 explicamos la necesidad de este valor, en este vamos a explicar el algoritmo que es utilizado para calcularle.

El MedianAlgorithm[20] tiene dos entradas, un vector con las cuentas actuales de cada canal y un segundo vector con la media de cuentas para cada canal. Este vector con cuentas medias es calculado durante un intervalo temporal en el que la presión atmosférica no fluctúa y además esta se corresponde al valor medio de presión para la estación. Además es deseable que durante dicho intervalo no ocurran eventos que pueden afectar la cantidad de partículas que llegan al instrumento. Este vector de valores medio es especificado en el archivo de configuración que previamente discutimos por lo que es fácilmente modificable. El algoritmo empieza calculando la desviación relativa sobre la media de cada canal, los canales con una desviación grande son descartados. Comparando la desviación de los canales restantes seleccionamos el mediano, de aquí el nombre del algoritmo. La desviación relativa del canal seleccionado es multiplicada por el sumatorio del vector de cuentas medias. El algoritmo devuelve el valor generado por la multiplicación. A continuación presentamos la implementación del algoritmo.

```
1 def medianAlgorithm(self, counts):
2     # Calculamos la desviación sobre la media.
3     r=[x/z for x,z in zip(counts, self.channel_avg)
4         # Descartamos los canales con desviación grande.
5         if z>0 and (x/z)>0.3 and (x/z)<10]
6     tet = numpy.median(r)
7     s0  = sum(self.channel_avg)
8     return s0*tet
```

Una vez calculado el valor global usando el MedianAlgorithm procedemos a calcular las dos correcciones. La primera es la corrección por presión. Como explicamos en el cápitulo 1 la presión atmosférica influye en el proceso de adquisición. Para realizar la corrección por presión hay dos factores que se toman en cuenta. El primero es la desviación de la presión actual respecto al valor medio de presión para la estación. El segundo es el coeficiente de correlación de los tubos respecto a la presión atmosférica. Este coeficiente ha sido calculado por el equipo de CaLMa, siendo el método utilizado desconocido para el autor de este trabajo. En la ecuación 3.1 podemos ver como se usan estos dos factores, donde N_0 es el valor sin corregir, N el valor corregido, β el coeficiente de corelación, P la presión actual y P_0 la presión media para la estación.

$$N = N_0 * exp(\beta * (P - P_0)) \tag{3.1}$$

La segunda corrección que realizaremos es la corrección por eficiencia. Esta es una corrección muy simple, consiste en aplicar un factor multiplicativo a la corrección por presión. Esta corrección se puede ver en la fórmula 3.2, donde N_0 es el valor de la corrección por presión, N el valor de la corrección por eficiencia y γ el factor de corrección.

$$N = N_0 * \gamma \tag{3.2}$$

Cambios en el entorno de la estación pueden causar cambios en la cantidad de eventos medidos, por ejemplo las precipitaciones de nieve pueden reducir la cantidad de partículas que llegan al instrumento. Una persona que analiza los datos con propósito científico puede equívocamente asociar el decremento con algún fenómeno físico, mientras que este es debido a un fenómeno técnico. Este problema es solventado con la corrección por eficiencia.

3.4.3. Base de datos

Como ya hemos explicado los datos son guardados en una base de datos para cuya gestión utilizamos *Sqlite3*. En la base de datos tenemos tres tablas que a continuación procedemos a explicar.

La primera tiene el nombre de `binTable`, nombre que hemos heredado de los primeros sistemas de adquisición rusos. En esta guardamos la información de las cuentas de cada minuto. Junto a las cuentas guardamos la fecha y hora en los que se realizó la medida, la lectura del barómetro y la lectura de las fuentes de alta tensión.

En la segunda tabla llamada `CALM_ori`, nombre definido por el NMDB, guardamos el valor global y sus correcciones. Junto a estos valores también guardamos el valor de la fecha y hora en los que se realizó la medida, y también el valor de la presión atmosférica.

En la última tabla guardamos los valores de los anchos de pulsos. Dado al gran número de pulsos, guardar la información de cada uno por separado no es práctico. Para ilustrar el problema pondremos de ejemplo la estación de CaLMa donde son procesados unos 4500 pulsos cada minuto, siendo esta una de las estaciones con menos eventos por minuto debido a su baja latitud. Para sobrevenir este problema construimos un histograma que guardamos en la base de datos, en forma de cadenas JSON[43]. Los histogramas se construyen con los datos de diez minutos, intervalo que marca la resolución de los datos de esta tabla.

3.5. DBUpdater

Este módulo software es el encargado de actualizar el contenido de la base de datos remota. El funcionamiento de este es muy simple y se puede ver en la figura 3.1. Empieza estableciendo la conexión con la base de datos remota. Si esta se establece se procede a leer la última entrada en la base de datos remota. Seguidamente el software calcula la diferencia entre las dos bases de datos, para este propósito usa la fecha y hora de la entrada leída de la base de datos remota. Finalmente el software selecciona las diferencias entre las dos bases de datos y las escribe en la remota. Es entonces cuando la ejecución de este *thread* termina. A fin de asegurar la sincronización en tiempo real se crea y lanza una instancia de este *thread* cada minuto.

Si la conexión con la máquina que alberga a la base de datos remota se pierde por un tiempo, cuando esta vuelva a restablecerse el `DBUpdater` sincronizará las dos bases de datos. Eventualmente si las diferencias entre las dos bases de datos son muy grandes la sincronización no ocurre de golpe, de esta manera evitamos sobrecargar al sistema.

3.6. SensorsManager

Este es el módulo software encargado de manejar los sensores. Por sensores nos referimos al barómetro, las fuentes de alta tensión o los termómetros si están presentes. A este módulo se le pasa la información referente a los sensores desde el archivo de configuración. De acuerdo con esta información este módulo invoca las funciones de configuración necesarias. Una vez realizada esta configuración inicial podemos empezar a leer la información de los sensores. Para este propósito este módulo exporta tres funciones, que se explican a continuación.

3.6.1. Presión atmosférica

Para leer el valor actual de la presión atmosférica este módulo nos ofrece la función `read_pressure`. Si en el archivo de configuración hemos especificado que no tenemos barómetro esta función devuelve un *-1*, así como en el caso de que se produzca algún error. Actualmente se soportan dos tipos de barómetros, el *BM35* y el *AP1*. La lógica para manejar estos dos barómetros se encuentra en los archivos `BM53Driver` y `AP1Driver` respectivamente. No explicaremos a fondo el funcionamiento de estos dos. Tan solo destacaremos que el *BM35* se comunica mediante un puerto serie y el *AP1* mediante tres señales digitales que son reloj, *strobe* y datos.

3.6.2. Fuentes de alta tensión

En el caso de las fuentes de alta tensión las tenemos de dos tipos, analógicas y digitales. Las digitales transmiten su información mediante un puerto serie. Las analógicas mediante una señal analógica entre 0V y 5V, que es proporcional a la tensión ofrecida por la fuente. La lógica para operar con las fuentes analógicas está en el archivo `HVPSDriver`, la lógica para las fuentes digitales no está implementada porque no hemos podido trabajar con estas. Eventualmente si se produce algún error se devuelve el valor de *-1*.

3.6.3. Temperatura

Como hemos comentado en muchas estaciones la temperatura ambiente es monitorizada también. Este no es el caso de CaLMa, razón por la que no hemos escrito ningunos drivers. Sin embargo el código está pensado para poder ser fácilmente ampliado en caso de necesidad.

3.7. Pruebas unitarias

Durante la realización de este trabajo hemos escrito una serie de test unitarios[44]. Los test unitarios son una técnica que permite comprobar el funcionamiento correcto de pequeños módulos o funciones. Las pruebas unitarias a diferencia de otras técnicas pueden ser usadas desde una etapa muy inicial del proyecto, incluso existen metodologías como *Desarrollo dirigido por pruebas* donde las pruebas son escritas primero y después es escrita la función que pasa la prueba. Recalcamos que en este trabajo no hemos seguido esta metodología.

Para que las pruebas unitarias sean aplicables es necesario que el código este bien estructurado. Un código con funciones pequeñas donde cada función tiene un propósito bien definido es fácil de testear. El simple hecho de poder escribir un test de una función indica que dicha función está bien estructurada. Escribir pruebas unitarias conlleva refactorizar el código, haciéndolo mejor.

Las pruebas unitarias son muy útiles y es recomendable que estas tengan una gran cobertura, pero no siempre es posible escribir pruebas unitarias para todas las funciones. Pongamos como ejemplo una función que lee una entrada desde una base de datos. Aunque la función este bien estructurada es difícil testearla por múltiples razones. Por ejemplo el valor devuelto depende de los datos presentes en la base de datos, suponiendo que esta está presente y está configurada correctamente. Es difícil escribir pruebas unitarias para este tipo de funciones, razón por la que existen las pruebas de integración.

Para realizar los test unitarios hemos utilizado *mocks*(objetos simulados). Estos son objetos que simulan el comportamiento de objetos reales de una forma controlada. Por ejemplo en nuestra aplicación hemos utilizado un *mock* para simular el comportamiento de los puertos serie. Veamos un pequeño ejemplo donde comprobamos la función `FPGASerialReader.run()`, función que es encargada de interpretar los datos que llegan por la UART de pulsos desde la FPGA.

```
1  def test_high_level_pulse(self):
2     port=MagicMock()
3     port.inWaiting.return_value=1
4     # Secuencia de bytes que representan un mensaje de
5     # ancho de pulso.
6     port.read.side_effect= \
7          ReturnSequence([['\x43'],['\x80'],['\xA0']],None)
8
9     reader=FPGASerialReader(port, None, [], [], [])
10    reader.run()
11    self.assertEqual(reader.shared_countsFromEvents[3],1)
12 #-------------------------------------------------------
13 class ReturnSequence(object):
14    def __init__(self, return_sequence, expired):
15       self.return_sequence = return_sequence
16       self.expired = expired
17    def __call__(self, *args):
18       if 0  < len(self.return_sequence):
19          return self.return_sequence.pop(0)
20       else:
21          return self.expired
```

El objeto `port` es el objeto que representa el puerto como vemos este es un *mock*. La función `inWaiting` que devuelve el número de datos disponibles es configurada para que siempre devuelva `1`. La función `read` es configurada para devolver múltiples valores en un orden concreto. Seguidamente es creado el objeto cuya función queremos probar pasándole la referencia del *mock*, una vez creado invocamos la función `run`. Finalmente comprobamos si el atributo `shared_countsFromEvents` tiene el valor esperado.

En el ejemplo anterior hicimos la comprobación sobre el valor de un atributo con la función `assertEqual()`. Los *mocks* ofrecen aún más funcionalidad, las funciones `assert_called_with()` y `assert_has_calls()` que permiten comprobar las llamadas que han sido realizadas sobre los *mocks*.

Junto a las pruebas unitarias hemos utilizado la biblioteca `coverage` que permite medir la cobertura de estas. Una mayor cobertura siempre es mejor, pero no debe convertirse en un objetivo principal. Como hemos dicho para ciertas funciones, como las que interactúan con una base de datos es difícil concebir una prueba unitaria. En la figura 3.4 presentamos el resumen de cobertura generado por `coverage`. Podemos ver que las pruebas para el `CountManager` no son muy extensas debido a que este es un módulo que interactúa mucho con las bases de datos. Sin embargo podemos ver que la cobertura del `DBUpdater` es del 100 %, esto es debido a que los test escritos para este módulo no son unitarios, esto son más bien test de integración. Durante el proceso de implementación surgió la necesidad de estos, razón por la que son incluidos en este trabajo.

Cobertura Pruebas unitarias

Coverage report: 75%

Module	*statements*	*missing*	*excluded*	*coverage*
BM35Driver.py	31	0	0	100%
CountsManager.py	124	63	0	49%
DBUpdater.py	30	0	0	100%
FPGASerialReader.py	112	6	0	95%
SensorsManager.py	60	20	0	67%
Total	**357**	**89**	**0**	**75%**

coverage.py v4.0a5

Figura 3.4: Cobertura Pruebas unitarias

Capítulo 4

Herramienta Web. *Back-End*

Fijándonos en el diseño preliminar de la figura 1.3, podemos ver que nuestra aplicación Web está dividida en *front-end* y *back-end*. Esta separación entre módulos es una técnica popular en diseño software. El *front-end* es el encargado de la capa de presentación, sobre este hablaremos más en el próximo capítulo. En este capítulo nos centraremos en explicar el *back-end*. Este es el responsable de procesar las peticiones provenientes del *front-end* y devolver le a este la información solicitada.

El *back-end* es implementado en *PHP*[18]. Este es un lenguaje diseñado para desarrollo Web y además es una elección muy popular. Hemos utilizado *ZendFramework*[31], este es un framework orientado al desarrollo de aplicaciones Web. Junto al framework hemos utilizado *Apigility*[28], herramienta que simplifica la creación y mantenimiento de APIs.

El *back-end* tiene un enfoque RPC o *Remote Procedure Call*. Este módulo exporta un conjunto de procedimientos que pueden ser invocados por el *front-end*. Estos procedimientos realizan tareas específicas sobre un recurso concreto. A lo largo de este trabajo nos referiremos a estos procedimientos como *servicios RPC*. Un aspecto oportuno destacar es que los servicios RPC son sin estado. Esto quiere decir que la respuesta a un procedimiento no se ve influenciada por eventos anteriores, esta tan sólo depende de los parámetros proporcionados.

Si nos volvemos a fijar en la figura 1.3 podemos ver que en el *back-end* existe la separación entre *Modelo* y *Controlador*. Esta separación sigue el patrón de diseño MVC[10]. Podemos ver que el componente de *Vista* no existe, en este caso el *front-end* es el componente de *Vista*. Todos los servicios RPC tienen su parte proporcional de *Modelo* y *Controlador*. La parte de *Modelo* es la encargada de gestionar la información contenida en la base de datos. La parte de *Controlador* es la encargada de gestionar los mensajes de petición y respuesta.

En este capítulo procederemos a explicar los servicios RPC, pero primero explicaremos algunos aspectos técnicos como la base de datos o el *ZendFramework*.

4.1. Protocolo de comunicación

Hemos explicado que entre el *back-end* y *front-end* existe una comunicación, para la que hemos elegido un enfoque RPC. Siendo la aplicación que desarrollamos una aplicación Web es de esperar que el protocolo utilizado para la comunicación entre los dos módulos sea *HTTP*. En este protocolo se puede diferenciar entre dos tipos de mensajes, mensajes de petición y de respuesta. En nuestra aplicación el *front-end* envia mensajes de petición al *back-end*. Los mensajes de petición especifican un campo URL,

este campo es muy importante porque especifica el servicio RPC que se desea invocar. A la URL también son anexados los parámetros necesarios para el procedimiento especificado. Los mensajes de respuesta serán discutidos en el capítulo dedicado el *front-end*. Los mensajes de petición también especifican un método, que puede ser *GET* o *POST* (o más cosas) dependiendo de si la operación es lectura o escritura. Todos los servicios RPC implementados por el *back-end* son servicios de lectura, a excepción de `nmdbMarkNull`.

4.2. ZendFramework y Apigility

Para la realización de este trabajo hemos utilizado *ZendFramework*, este es un framework orientado al desarrollo de aplicaciones y servicios Web. Basado en *PHP 5.3+* el framework sigue un diseño orientado a objetos. Esta enfocado en la creación de aplicaciones según el patrón MVC, ofrece abstracciones para las bases de datos, autenticación y validación de parámetros. El framework también disfruta de una amplia comunidad de usuarios. Todas estas ventajas nombradas anteriormente son la razón para elegir este framework en nuestro trabajo.

A la hora de abordar un diseño desde cero, el framework ofrece la aplicación esqueleto. Esta aplicación consiste del código mínimo necesario para construir una aplicación usando *ZendFramework*, no tiene ninguna funcionalidad y está pensada para ser extendida. En este trabajo hemos empezado con esta aplicación esqueleto.

Apigility es una herramienta creada por el equipo responsable de *ZendFramework*. Esta puede utilizarse sin necesidad del framework, sin embargo se integra muy bien con este. El obejetivo principal de esta es facilitar la creación y mantenimiento de aplicaciones Web. En nuestro caso nos ha ayudado a crear los servicios RPC de nuestra aplicación. La herramienta ofrece un entorno gráfico accesible desde un navegador Web, que es muy fácil e intuitivo de utilizar.

4.3. Bases de datos

Tal y como hemos explicado al principio del capítulo la función del *back-end* es procesar las peticiones del *front-end* y responderle con la información solicitada. Esta información es guardada en dos bases de datos. Para la gestión de las bases de datos utilizamos *MySql*[1]. En la figura 4.1 podemos ver el esquema de las tablas con las que vamos a trabajar.

En la tabla `binTable` guardamos la información de las cuentas de cada canal. Junto a esta información guardamos las lecturas del barómetro, las fuentes de alta tensión y la fecha y hora actuales. La resolución de los datos es de un minuto.

En la tabla `CALM_ori` guardamos el valor global y las correcciones sobre este. La lectura de presión atmosférica también es guardada. La resolución de estos datos es también de un minuto.

En la tabla `CALM_rev` guardamos la revisión de los datos de `CALM_ori`. Vemos que la tabla tiene el mismo esquema con dos campos adicionales. En estos dos campos se guardan la fecha de última modificación y la versión. Los datos en esta tabla son introducidos por un operario. Cuando este encuentra un dato corrupto en los datos originales, puede crear una entrada en esta tabla para señalarlo.

La tabla `binTable` está almacenada en una base de datos con el nombre `nmdadb`. Las tablas `CALM_ori` y `CALM_rev` están en otra base de datos con el nombre `nmdb`. En este documento hablaremos a nivel de tablas, muchas veces sin tener en cuenta la

separación en diferentes bases de datos. En este punto destacamos esta separación para no crear confusión al lector a la hora de contrastar el código fuente con este documento.

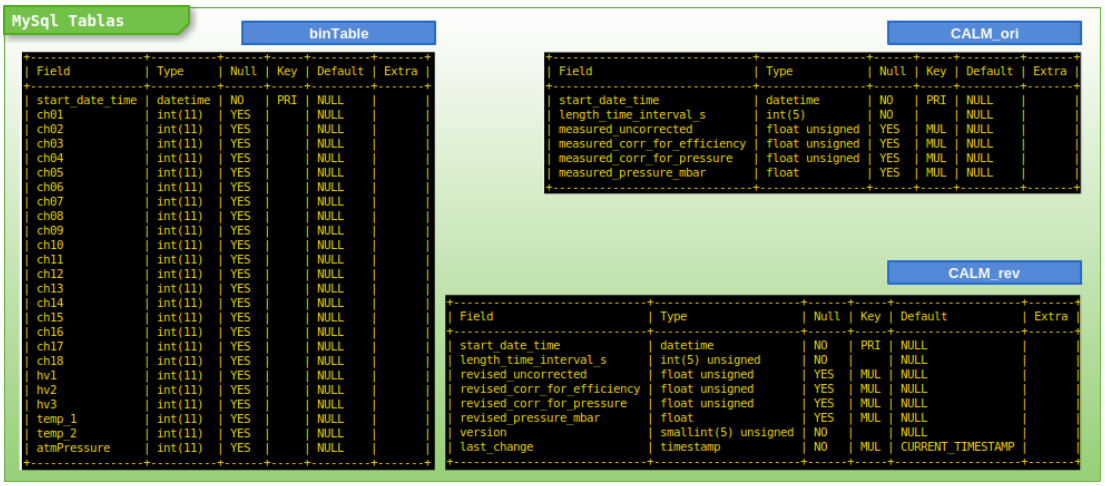

Figura 4.1: Esquema de las tablas.

Es interesante destacar que en las tres tablas un existe índice. En los tres casos el elemento indexado es el campo `start_date_time`. Este índice es muy útil para la mayoría de consultas que realizamos sobre los datos.

ZendFramework ofrece abstracciones para las bases de datos que facilitan el trabajo con estas. A continuación presentamos un pequeño ejemplo de cómo hacer uso de estas facilidades. `Zend\Db\Adapter\AdapterInterface` es la abstracción que representa un conector a una base de datos. En este ejemplo la variable `$this->adapter` es una instancia de esta clase. La variable `resultSet` instancia de la clase `Zend\Db\ResultSet\ResultSet` es un iterable que contiene los datos devueltos por la consulta ejecutada.

```
1  use Zend\Db\ResultSet\ResultSet;
2  use Zend\Db\Adapter\AdapterInterface;
3  //------------------------------------------------
4  public function interval($start,$finish){
5     $sql = "SELECT * FROM binTable WHERE start_date_time "\
6             ."between '".$start."' and '".$finish."';";
7     $result = $this->adapter->query($sql)->execute();
8
9     $resultSet = new ResultSet;
10    $resultSet->initialize($result);
11    return $resultSet;}
```

4.4. Servicios RPC

En esta sección procedemos a explicar todos los servicios RPC que nuestro *back-end* ofrece. Para cada servicio RPC explicamos la funcionalidad, la URL que le identifica, los parámetros aceptados, la consulta SQL utilizada y el propósito de este.

4.4.1. nmdbOriginalRaw

Este servicio RPC devuelve los datos de la tabla `CALM_ori` en un intervalo determinado. Tal y como explicamos en esta tabla son guardados los datos globales de la estación, las dos correcciones de estos y los valores de presión atmosférica. Este servicio devuelve los datos tal y como están en la base de datos.

La URL que identifica a este servicio tiene el siguiente formato.

```
/nmdb/original/raw[/:start][/:finish]
```

Como vemos el servicio acepta dos parámetros, `start` y `finish`. Los parámetros delimitan el intervalo de datos devueltos.

A continuación podemos ver la consulta SQL utilizada para obtener los datos, dado a la simplicidad de esta no procederemos a discutirla más a fondo.

```
SELECT * FROM CALM_ori
WHERE start_date_time BETWEEN '.$start.'AND '.$finish'
```

Los datos devueltos por este servicio RPC son utilizados para construir un gráfico de líneas.

4.4.2. nmdbOriginalGroup

Este servicio RPC es similar al anteriormente descrito en el sentido de que devuelve los datos de la tabla `CALM_ori` en un intervalo determinado. La diferencia radica en que este no devuelve los datos en crudo, sino que los datos son agrupados y después se devuelven valores descriptivos para cada grupo.

A continuación podemos ver el formato de la URL que identifica a este servicio.

```
/nmdb/original/group[/:start][/:finish][/:points]
```

Los dos parámetros `start` y `finish`, al igual que en el servicio anterior, delimitan el intervalo de datos. En este servicio estos dos parámetros pueden tener el valor `all`, de ser así son utilizados todos los datos presentes en la base de datos. El parámetro `points` especifica el número de grupos en los que serán agrupados nuestros datos.

Seguidamente podemos ver parte de la consulta SQL utilizada en este servicio. Esta parte es la encargada de agrupar nuestros datos.

```
(SELECT CALM_ori.*,
(UNIX_TIMESTAMP(start_date_time) DIV (.$interval.)) AS timekey
FROM CALM_ori WHERE start_date_time BETWEEN '.$start.'AND '.$finish.')AS
t1 GROUP BY timekey
```

Como podemos ver son seleccionados los datos presentes en el intervalo definido por `$start` y `$finish`. Junto a los datos presentes en la tabla `CALM_ori` es calculado el campo `timekey`. Este campo es el utilizado para formar los grupos. El `timekey` es calculado utilizando el valor de la variable `$interval`, que a su vez se calcula de la siguiente forma:

```
$interval =round(($finish-$start)/($points));
```

Una vez formados los grupos, para cada uno de ellos se calcula el máximo, mínimo, *open* y *close*. El *open* y *close* son la media más la desviación típica y la media menos la desviación típica respectivamente. Estos cuatro valores son calculados para el

valor global, la corrección por presión y la corrección por eficiencia. Para la presión atmosférica tan sólo es calculado el valor medio. Estos datos son utilizados para construir un gráfico *Candlestick*. Sobre este tipo de gráfico hablaremos en el siguiente capítulo dedicado al *front-end*.

4.4.3. nmdbRevisedRaw

Este servicio es muy similar al `nmdbOriginalRaw`, la diferencia está en que este devuelve los datos revisados. Para este propósito son utilizados los datos de las tablas `CALM_ori` y `CALM_rev`. La URL utilizada para identificar este servicio tiene el siguiente formato.

```
/nmdb/revised/raw[/:start][/:finish]
```

Los dos parámetros aceptados delimitan el intervalo de datos.

En la tabla `CALM_rev` son guardados los datos revisados, pero no son guardados todos, tan sólo las entradas que han sido modificadas. Por esta razón este servicio también hace uso de `CALM_ori` para devolvernos la información completa. A continuación podemos ver parte de la consulta SQL utilizada para obtener los datos revisados.

```
SELECT ori.start_date_time,
CASE WHEN rev.start_date_time IS NULL THEN ori.measured_uncorrected ELSE
rev.revised_uncorrected END AS uncorrected
FROM CALM_ori ori LEFT JOIN CALM_rev rev ON ori.start_date_time =
rev.start_date_time
```

La consulta empieza haciendo un `LEFT JOIN` sobre el campo `start_date_time` de las tablas `CALM_ori` y `CALM_rev`. El `LEFT JOIN` devuelve todas las entradas de `CALM_ori`, con las entradas coincidentes de `CALM_rev`. La parte correspondiente al `CALM_rev` es dejada a `NULL` si no existe coincidencia. A partir de la tabla temporal creada por el `LEFT JOIN` son seleccionados los datos de `CALM_rev` si estos están presentes, en caso contario son utilizados lo datos de `CALM_ori`. Los datos devueltos por este servicio se utilizan para crear un gráfico de líneas.

4.4.4. nmdbRevisedGroup

Este servicio es muy similar al `nmdbOriginalGroup`, pero este hace uso de los datos revisados. La URL que identifica este servicio tiene el siguiente formato.

```
/nmdb/revised/group[/:start][/:finish][/:points]
```

Los parámetros `start` y `finish` delimitan el intervalo de datos. Eventualmente estos dos parámetros pueden tener el valor de `all`, de esta manera son usados todos los datos presentes en la base de datos. El tercer parámetro, `points`, especifica el número de grupos en los que queremos agrupar nuestros datos.

Para recolectar los datos necesarios la consulta utilizada es una combinación de las utilizadas en los servicios `nmdbOriginalGroup` y `nmdbRevisedRaw`. Primero se realiza un `LEFT JOIN` entre las tablas `CALM_ori` y `CALM_rev` para obtener el conjunto de datos revisados. Seguidamente se agrupan los datos de este conjunto y para cada grupo se calculan los valores máximo, mínimo, *open* y *close*. Estos datos son utilizados para construir un gráfico *Candlestick*.

4.4.5. nmdbMarkNull

Este servicio es el más peculiar de todos, ya que los demás servicios son de consulta, mientras que este es de modificación. Uno de los propósitos de nuestra aplicación Web es permitir visualizar los datos de la estación y al visualizar un dato corrupto poder marcarlo. Para satisfacer este requisito nuestro *back-end* implementa este servicio que permite marcar como nulo un dato en el conjunto de datos revisados. De esta manera pueden ser descartados todos los datos no deseados.

A continuación podemos ver la URL utilizada para identificar este servicio.

```
/nmdb/marknull
```

Como podemos ver este servicio no necesita parámetros. Este servicio acepta un método `POST`. Los mensajes con método `POST` siempre van acompañados de un campo de datos. Este campo de datos contiene la información del dato que queremos marcar como nulo. Eventualmente este campo puede contener un array, de esta forma podemos marcar más de un dato con sólo una llamada al servicio.

La consulta SQL utilizada por este servicio es muy simple, esta realiza un `INSERT` en la tabla `CALM_rev`. Si el dato ya está presente tan sólo incrementamos la versión actual y modificamos la fecha de última modificación. A continuación podemos ver la consulta que estamos discutiendo.

```
                    INSERT INTO CALM_rev
   (start_date_time, revised_uncorrected, revised_corr_for_efficiency,
revised_corr_for_pressure, revised_pressure_mbar, version, last_change )
        VALUES ('.$start_date_time.',null,null,null,null,1,now())
      on DUPLICATE KEY UPDATE version=version+1, last_change=now()
```

4.4.6. nmdadbRawData

Este servicio tiene como propósito devolver lo datos crudos de la tabla `binTable`. La URL utilizada para identificar este servicio tiene el siguiente formato.

```
/nmdadb/rawdata[/:start][/:finish]
```

Los parámetros `start` y `finish` delimitan el intervalo de datos. Para recolectar los datos crudos de la tabla `binTable` este servicio usa una consulta muy simple, que se muestra a continuación.

```
                    SELECT * FROM binTable
   WHERE start_date_time between '.$start.'and '.$finish.'
```

Los datos devueltos por este servicio son utilizados para construir un gráfico de líneas.

4.4.7. nmdadbChannelStats

Este servicio tiene como propósito devolver estadísticas descriptivas para cada canal de la estación. Para este propósito se usan los datos de la tabla `binTable`. Para cada canal son calculados el valor medio, desviación típica, mínimo y máximo. A continuación podemos ver la URL utilizada para identificar este servicio.

```
/nmdb/channel/stats[/:start][/:finish]
```

Los parámetros `start` y `finish` delimitan el intervalo de datos. Eventualmente estos dos parámetros pueden tener el valor `default`, en este caso el intervalo es el último mes con datos.

A continuación podemos ver parte de la consulta utilizada para recolectar los datos devueltos por este servicio.

```
select 'ch01' as n1, ch01, avg(ch01) as avg_ch01, std(ch01) as std_ch01,
             max(ch01) as max_ch01, min(ch01) as min_ch01
from(select * from binTable where start_date_time between '.$start.'and
            '.$finish.'order by start_date_time desc)as t1
```

Los datos generados por este servicio son utilizados para crear un gráfico *Candlestick* y también son presentados en una tabla.

4.4.8. `nmdadbChannelHistogram`

Este servicio devuelve un histograma de la distribución de cuentas para los 18 canales. Para cumplir con dicho propósito este servicio utiliza los datos de la tabla `binTable`. A continuación podemos ver la URL utilizada para identificar este servicio.

```
/nmdb/channel/histogram[/:start][/:finish]
```

Al igual que en los otros servicio los parámetros `start` y `finish` delimitan el intervalo de datos usados para construir el histograma. Eventualmente estos dos parámetros pueden tener el valor de `default` y en este caso son utilizados los datos del último mes.

Seguidamente podemos ver la consulta utilizada para construir el histograma devuelto por este servicio.

```
select chh as num, count(chh) as val from (select ch div 5 as chh from
 (select .$channel. as ch from binTable where start_date_time between
  '.$start.'and '.$finish.') as t1) as t2 group by chh order by num
```

La consulta calcula el histograma para un único canal, especificado por la variable `$channel`. Para construir el histograma de los 18 canales esta consulta es invocada múltiples veces. En esta consulta tenemos que agrupar por el número de cuentas, campo que no está indexado. Esto hace que esta tarde bastante más tiempo que las otras, por lo tanto no es recomendable invocarla con mucha frecuencia sobre conjuntos muy grandes.

Capítulo 5

Herramienta Web. *Front-End*

Nuestra aplicación Web está dividida en *back-end* y *front-end*. En el capítulo anterior se describió el *back-end*. El propósito de este capítulo es describir el *front-end*, que es el encargado de la capa de presentación.

El *front-end* se implementa en *JavaScript*[42]. Este es un lenguaje de programación soportado por la mayoría de navegadores, que permite dotar de funcionalidad extra a las páginas Web. Actualmente el uso de este lenguaje está tan extendido y avanzado que permite crear aplicaciones enteras para navegadores Web. Existen múltiples frameworks escritos en *JavaScript* que facilitan la creación de aplicaciones Web. En este trabajo utilizamos dos: *Sencha ExtJs*[22] y *HighStock*[3].

Sencha ExtJs es un framework orientado a la creación de aplicaciones Web interactivas. Se trata de un framework de propósito general que ofrece abstracciones para gestionar nuestros datos, arquitectura MVC, componentes gráficos de control y otros.

HighStock es un framework con un propósito específico, que es la creación de gráficos. Podemos elegir entre diferentes tipos de gráficos, que pueden ser: lineas, barras, áreas, *candlestick* y otros. Los gráficos generados son altamente interactivos lo que permite ocultar series, navegar, realizar *zoom* y mucho más.

Empezaremos este capítulo aclarando los aspectos técnicos relacionados con estos dos frameworks. Explicaremos como usarlos, que funcionalidad nos proporcionan, consideraciones que debemos tener en cuenta.

Como podemos ver en la figura 5 nuestro *front-end* está basado en el patrón de diseño *modelo-vista-controlador*[10]. En la segunda parte de este capítulo hablaremos más a fondo sobre la división impuesta por este patrón de diseño.

En nuestra aplicación también hacemos una división funcional, diferenciamos entre tres módulos que son `Spike`, `SpikeRevised` y `ChannelStats`. En las secciones finales de este capítulo hablaremos sobre estos módulos.

5.1. Sencha ExtJs

El propósito de esta sección es explicar alguno de los aspectos básicos del framework. Empezaremos explicando como crear una aplicación básica con *ExtJs*. En la mayoría de los casos existe un único documento HTML[40] que contiene toda la aplicación. En él tenemos que cargar dos *scripts* de la siguiente forma.

```
<script type="text/javascript"src="extjs/ext-all-debug.js"></script>

        <script type="text/javascript"src="app.js"></script>
```

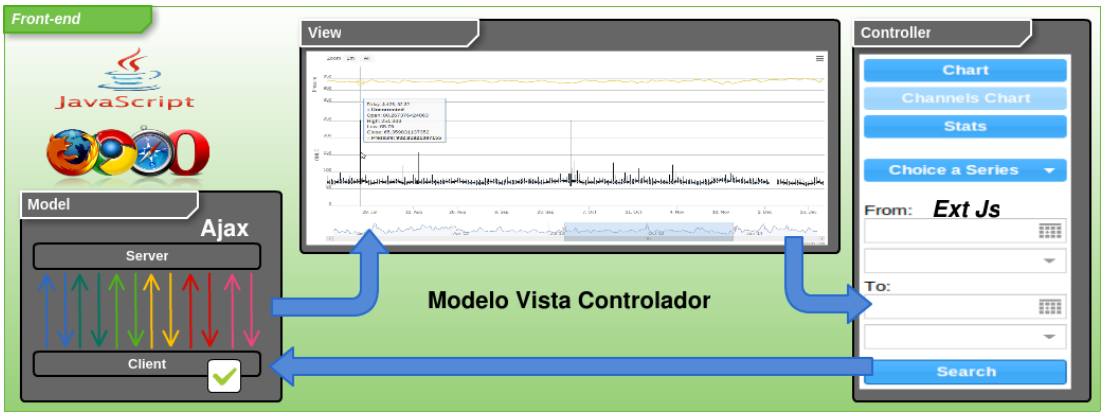

Figura 5.1: *Front-end*. Patrón MVC.

El primer *script* contiene el framework que queremos utilizar. Es conveniente destacar que esta es una versión concebida para el proceso de desarrollo. Para la versión final es conveniente usar el `ext-all.js`, que es una versión comprimida del mismo *script*.

El segundo *script* es el que contiene la lógica de nuestra aplicación. A continuación podemos ver un ejemplo del código mínimo que este *script* debe contener. El código presentado se explicará a fondo.

```
1  Ext.application({
2      name: 'HelloExt',
3      launch: function() {
4          Ext.create('Ext.container.Viewport', {
5              layout: 'fit',
6                  items: [{
7                      title: 'Hello Ext',
8                      html : 'Hello! Welcome to Ext JS.'}]
9          });
10     }
11 });
```

En la primera línea hacemos uso del singleton `Ext`. Este es un objeto que encapsula todas las clases y métodos proporcionados por el framework. La función utilizada `Ext.application` carga e inicializa una instancia de la clase `Ext.app.Application`. Esta clase representa una aplicación *ExtJs single-page*. La llamada a esta función crea la variable global `MyApp`, que debe contener todas las clases e instancias de nuestra aplicación.

En la segunda línea declaramos el nombre de nuestra aplicación. Seguidamente definimos la función `launch`. Esta función es ejecuta cuando se lanza la aplicación. La primera función invocada es `Ext.create` que crea una instancia de la clase proporcionada, en este caso `Ext.container.Viewport`. `Viewport` es un *contenedor* que representa el área de aplicación y puede haber tan sólo uno por aplicación.

La interfaz de usuario en una aplicación *ExtJs* se compone de *componentes*. Un *contenedor* es un *componente* especial que contiene otros *componentes*. En la figura 5.2 podemos ver un ejemplo que ilustra esta jerarquía.

El `Viewport` es el *contenedor* que gestiona el área de representación del navegador Web. Los *compenentes* de un *contenedor* se especifican en el campo `items` que es una lista. En el ejemplo presentado tan sólo tenemos un *componente*, pero pueden añadirse más.

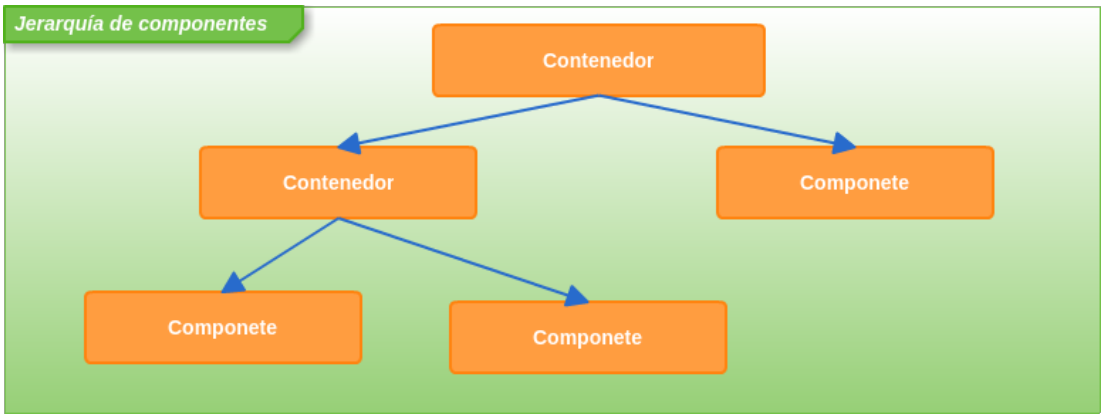

Figura 5.2: *Sencha ExtJs.* Jerarquía de componentes.

Fijándonos en el código de ejemplo podemos ver que antes de definir el campo `items` definimos el campo `layout`. El `layout` especifica la forma en la que se posicionan y ajustan los *componentes* hijos dentro del padre. En este caso el `layout` especificado es `'fit'`, donde un único hijo ocupa todo el espacio del padre.

5.1.1. Component Manager

ExtJs ofrece el singleton `Ext.ComponentManager` que provee un registro con todos los componentes. Este facilita la referencia a elementos desde cualquier punto del código. El propósito de esta sección es explicar como hacer uso de esta facilidad.

Empezaremos especificando un requisito que los componentes deben cumplir, estos deben definir el atributo `itemId`. Este atributo es el identificador del componente. El ámbito que tiene este identificador es local al contenedor que contiene el componente. Esto hace que los identificadores no tengan que ser únicos para toda la aplicación.

Es el singleton `Ext.ComponentQuery` el que permite realizar búsquedas de componentes en función del `itemId`. Concretamente es `Ext.ComponentQuery.query()` el método que permite realizar las búsquedas. Este método acepta dos parámetros. El primer parámetro, `selector`, es un String y especifica el `itemId` que queremos buscar. El segundo parámetro, `root` es opcional y especifica el contenedor dentro de cual queremos hacer la búsqueda. Si el segundo parámetro se omite la búsqueda se realizará entre todos los componentes. La función devuelve un array con todos los componentes que reúnen las condiciones, pudiendo ser un array vacío. Eventualmente podemos hacer búsquedas más complejas, que nos permiten buscar por tipo, atributos y mucho más. En este trabajo tan sólo hemos utilizado consultas simples.

La clase `Ext.container.Container` implementa las funciones `down()` y `query()`. Estas dos realizan una llamada a `Ext.ComponentQuery.query()` teniéndose a si mismo como parámetro `root`. Esto les permite hacer una búsqueda entre sus hijos.

También existe el método `up()` implementado por `Ext.Component`. Este permite hacer una búsqueda similar a las anteriormente descritas. En este caso se navega hacia arriba en la jerarquía de componentes y se busca por componentes que reúnan los criterios. Esta función puede ser invocada sin ningún parámetro, caso en el que devuelve el padre inmediato del componente.

Es conveniente citar el parámetro `id` y la función `Ext.getCmp()`. Estos dos ofrecen una funcionalidad parecida a la anteriormente explicada, pero no deben ser usados con ese propósito. Estan considerados como obsoletos y pueden dar lugar a colisiones.

5.1.2. Layouts

El propósito de esta sección es explicar los `layouts` utilizados en este trabajo. Tal y como explicamos anteriormente, el `layout` especifica la forma en la que se posicionan y ajustan los componetes hijos dentro del padre. Tan sólo explicaremos la funcionalidad de estos sin centrarnos en el uso que les hemos dado, este será explicado en las secciones venideras cuando proceda. En la figura 5.3 podemos ver una representación de los `layouts` que describiremos a continuación.

`Absolute` Los *componentes* hijo son posicionados mediante el uso de coordenadas `X,Y`. Las dimensiones de estos también se definen de forma estática mediante el uso de dos atributos, `height` and `width`. La posición y tamaño de los componentes se puede cambiar mediante el uso de las funciones `setPosition()` y `setSize()`.

`Accordion` Este layout se asemeja a un acordeón. Los *componentes* hijo pueden ser expandidos y colapsados, teniendo tan sólo uno expandido a la vez. Los *componentes* son ordenados en vertical ocupando todo el espacio disponible. Las funciones `expand` y `collapse` permiten expandir y colapsar los paneles. También están presentes multitud de manejadores de eventos que podemos sobrescribir. En nuestro caso hacemos uso del `beforeExpand` que se dispara antes de expandir un *componente*.

`Border` Este layout permite tener hasta cinco *componentes* dispuestos de la manera que podemos ver en la figura 5.3. Los componentes hijos deben especificar el atributo `region` que determina la posición que tendrán. El atributo puede tomar los siguientes valores: `[north, east, south, west, center]`. No es necesario que estén presentes los cinco hijos, podemos omitir los que no sean necesarios.

`Card` Este layout maneja multiples *componentes* hijo. Los hijos ocupan todo el espacio disponible, por lo que se solapan entre ellos. Esto hace que solamente uno sea visible a la vez. El layout asemeja una baraja de cartas donde solamente una carta puede estar en la parte superior. La función que permite hacer visible un hijo es `setActiveItem()`. Esta función acepta el *componente*, el `itemId` o el índice de este.

`Fit` Este es un layout muy simple, que tan sólo acepta un hijo, y este ocupa todo el espacio que el padre ofrece. El *componente* hijo se expande automáticamente para ajustarse al tamaño del padre.

`HBox y VBox` Estos son dos layouts diferentes, pero muy parecidos. `HBox` organiza sus elementos hijos de forma horizontal a lo largo del *componente* padre. `VBox` hace lo mismo, pero la organización es de forma vertical. Los componentes hijos pueden especificar el atributo `flex`, que determina como será repartido el espacio disponible entre los diferentes hijos. Para ilustrar el funcionamiento del `flex` expondremos un ejemplo. Teniendo dos *componentes* hijo con `1` y `2` de `flex` respectivamente, el primero ocupará 1/3 y el segundo 2/3 del espacio total.

5.1.3. Lazy instantiation

La inicialización perezosa es una técnica que consiste en retrasar la creación de recursos hasta el momento en el que sea necesario su uso. Como hemos visto en la sección dedicada a los layouts en muchos casos los componentes no son visibles. Tener que crear todos los componentes al inicializar la aplicación no es muy eficiente, dado que tan sólo unos pocos son visibles. Crear tan sólo los componentes visibles según se

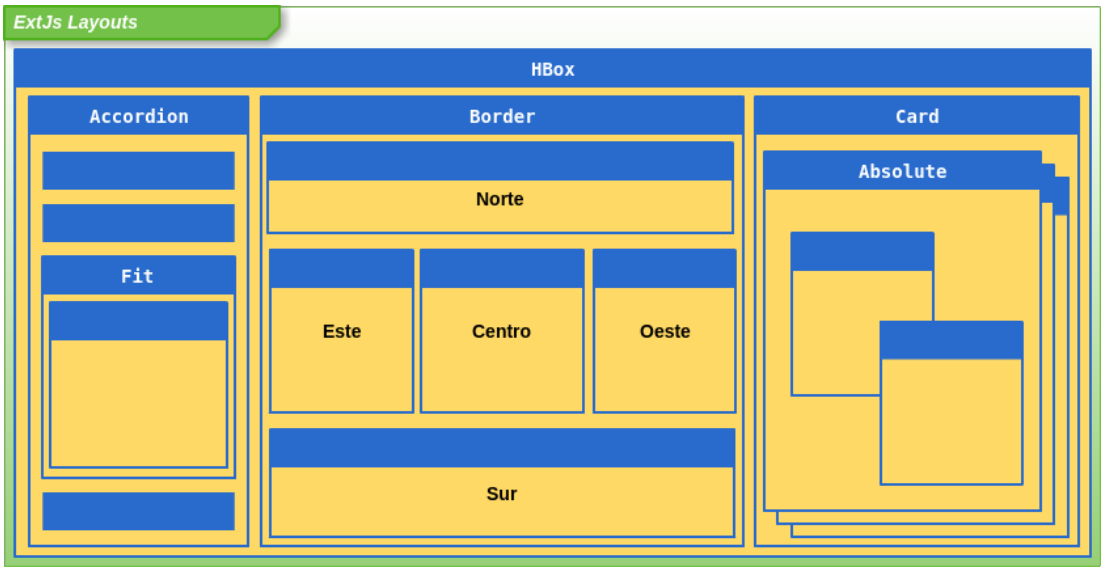

Figura 5.3: *Sencha ExtJs*. Layouts.

vayan necesitando supone un incremento en el rendimiento. En páginas pequeñas con pocos componentes el aumento no es apreciable, sin embargo en páginas que manejan multitud de componentes el incremento puede ser drástico.

Para implementar esta inicialización perezosa *ExtJs* se basa en la jerarquía de componentes previamente explicada. Cuando un componente es creado también se crean todos los componentes hijos que deben ser visibles en ese momento. Tal y como explicamos los componentes hijos se definen en el campo `items`. A continuación podemos ver un pequeño ejemplo.

```
1 items:[
2     Ext.create('Ext.form.field.Text',{
3         fieldLabel:'Foo'}),
4     Ext.create('Ext.form.field.Text', {
5         fieldLabel: 'Bar'})]
```

El ejemplo presentado anteriormente no habilita la inicialización perezosa. Para este propósito tenemos que evitar el uso de la función `Ext.create()`. Tenemos que especificar los componentes hijos como objetos de configuración. A continuación se presenta un ejemplo.

```
1 items: [{
2     xtype: 'textfield',
3     fieldLabel: 'Foo'},
4 {
5     xtype: 'textfield',
6     fieldLabel: 'Bar'}]
```

La diferencia que podemos apreciar entre los dos ejemplos es el atributo `xtype`. Este especifica la clase que tendrá el componente. Como podemos ver no se especifica el nombre completo de la clase. `xtype` acepta una serie de atajos para referenciar a las clases. En nuestro caso la clase `Ext.form.field.Text` es referenciada por el `xtype:'textfield'`.

La lista completa de `xtypes` aceptados se puede consultar en la documentación de *ExtJs*[24]. Eventualmente podemos extender esta lista declarando nuestros propios

`xtypes`. Esto se hace especificando el campo `xtype` a la hora de definir nuestra clase. A continuación presentamos un pequeño ejemplo.

```
Ext.define('MyApp.PressMeButton', {
    extend: 'Ext.button.Button',
    xtype: 'pressmebutton',
    text: 'Press Me'});
```

5.1.4. Events

Las clases y componentes de *ExtJs* disparan multitud de eventos a lo largo de su ciclo de vida. Los eventos se disparan cuando algo interesante le ocurre a nuestro componente. Un ejemplo es el evento `afterrender` que se dispara después de mostrar en pantalla el componente.

Los eventos son muy interesantes porque podemos definir funciones, `listeners`, que se ejecutan cuando se producen dichos eventos. A continuación presentamos un ejemplo en el que podemos ver cómo manejar el evento `click`, que se dispara al hacer *click* sobre un componente.

```
Ext.create('Ext.Button',{
    text: 'Click Me',
    renderTo: Ext.getBody(),
    listeners:{
        click: function(){
            Ext.Msg.alert('I was clicked!');}
    }
});
```

ExtJs también permite configurar eventos propios. Estos se declaran como si fuesen eventos normales. La peculiaridad está en que somos nosotros los que debemos dispararlos haciendo uso de la función `fireEvent`. Esta acepta como parámetros el evento que queremos disparar y todos los parámetros que la función encargada de este necesite. A continuación podemos ver un pequeño ejemplo de como declarar y disparar un evento propio, que tiene como nombre `myEvent`.

```
var button = Ext.create('Ext.Button',{
    text: 'Custom Event',
    renderTo: Ext.getBody(),
    listeners:{
        myEvent: function(number){
            Ext.Msg.alert('Custom event fired, number: '+number);}
    }
});
button.fireEvent('myEvent', 42);
```

5.1.5. Componentes

ExtJs ofrece multitud de componentes, que son subclases del `Ext.Component`, cada uno con un propósito diferente. El objetivo de esta sección es describir los componentes usados en este trabajo. Nos centraremos en los aspectos técnicos sin especificar el uso concreto que les hemos dado. Este será explicado en secciones futuras cuando proceda.

`Ext.button.Button xtype:button` Este componente permite crear un botón. El propósito de un botón es ofrecer un medio fácil al usuario para interactuar con

la aplicación. Los botones que *ExtJs* ofrece son altamente configurables, tanto funcionalmente como estéticamente. Podemos tener botones normales, botones *toggle*, botones que despliegan menús y mucho más. Para dar funcionalidad a estos botones tenemos que escuchar los eventos que estos disparan. Los eventos más comunes son `click`, `toggle`, `mouseover` y `mouseout`.

`Ext.form.field.Date xtype:datefield` Este componente proporciona un campo de entrada de fecha. Es posible especificar el formato deseado en el que introducir la fecha, y si este formato no es respetado el campo es subrayado en color rojo indicando al usuario la inconsistencia. Eventualmente este campo puede desplegar un selector de fechas que facilita el proceso de introducir la fecha deseada. Al igual que los demás componentes este dispara multitud de eventos que podemos manejar.

`Ext.form.field.Time xtype:timefield` Este componente es muy similar al anteriormente descrito, y proporciona un campo de entrada de tiempo. También acepta una configuración del formato deseado y advierte al usuario cuando este no es respetado. El componente puede desplegar un selector de tiempo que facilita la tarea de introducir el valor deseado.

`Ext.form.Label xtype:label` Este componente permite generar una etiqueta de texto. Este componente es muy similar al tag HTML <label>. Este componente permite configurar múltiples aspectos y manejar múltiples eventos.

`Ext.panel.Panel xtype:panel` Este componente es una parte fundamental en la creación de aplicaciones con *ExtJs*. Esta pensado para ser utilizado como un contenedor, contener otros componentes. Este componente es altamente configurable pero el aspecto más importante es el `layout` que determina como se posicionan los componentes hijos.

`Ext.tab.Panel xtype:tabpanel` Este componente es una extensión del componente anterior. Este ofrece múltiples pestañas en la que organizar los componentes hijos. Esto permite incluir una cantidad mayor de contenido en el mismo espacio. La barra de pestañas que el componente ofrece es altamente configurable.

`Ext.grid.Panel xtype:gridpanel` Este componente es un contenedor capaz de presentar una tabla. Las tablas son altamente interactivas, estas permiten al usuario ordenar los datos, hacer búsquedas, eliminar datos, añadir datos y mucho más.

5.1.6. History stack

La aplicación Web que estamos desarrollando es una aplicación *single-page*. Este modelo rompe con el patrón tradicional de navegación por el historial usando los botones *Forward/Back*. Esto presenta un problema de usabilidad cuando el usuario utiliza el botón de navegación hacia atrás, ya que lo esperado es volver al estado anterior de la aplicacióin. Sin embargo se carga la página anterior del historial y nuestra aplicación *single-page* es descartada.

La solución que la mayoría de aplicaciones *single-page* han adoptado se basa en el *Fragment Identifier* o *hash*, nombre que recibe porque va precedido del símbolo #. El *hash* es una parte opcional de la URL que históricamente se utilizaba para navegar por documentos largos. Un cambio en el *hash* hace que se muestren diferentes partes del documento.

Actualmente el *hash* es usado por aplicaciones *single-page* para manipular la pila de historial. El *hash* es cambiado acuerdo al estado de la aplicación. Esto hace que los cambios en la aplicación sean registrados en la pila de historial.

Al pulsar el botón de navegación hacia atrás el *hash* de la URL cambia a su estado anterior. Nuestra aplicación debe ser capaz de detectar los cambios en el *hash* y actuar acuerdo a estos. De esta manera podemos preservar la funcionalidad de los botones de navegación *Forward/Back*.

Para implementar la funcionalidad anteriormente descrita *ExtJs* ofrece el singleton `Ext.util.History`. Para inicializar este módulo tenemos que invocar la función `Ext.History.init()`. Una vez inicializado este módulo podemos hacer cambios en el *hash* invocando la función `Ext.History.add()`. Esta función acepta como parámetro el valor del *hash*.

Para detectar y actual acorde a los cambios del *hash* tenemos que inicializar un `listener`. El evento que vamos a capturar es `change`. A continuación podemos ver un pequeño código de como inicializar el `listener`.

```
1  Ext.History.on('change', function(hash){
2      switch(hash){
3          case: 'Foo':
4              //Actuar acuerdo al hash Foo
5              break;
6          case: 'Bar'
7              //Actuar acuerdo al hash Bar
8              break;
9      };
10 });
```

El propósito de esta sección ha sido explicar el problema que surge con el historial de navegación y como prevenirlo haciendo uso del `Ext.History`. Del uso concreto que hemos hecho de este módulo hablaremos en secciones futuras de este capítulo.

5.2. HighStock

El propósito de esta sección es explicar alguno de los aspectos básicos del framework. Este permite añadir gráficos interactivos de forma simple a páginas o aplicaciones Web. Es conveniente destacar la diferencia entre *HighCharts* y *HighStock*. El primero tiene un propósito más general, dado que ofrece mayor variedad de gráficos. El segundo tiene un propósito más específico, ya que está enfocado a la representación gráfica de valores que evolucionan respecto al tiempo. A continuación presentaremos el código mínimo necesario para crear un gráfico con *HighStock*. Basándose en este código explicaremos los aspectos técnicos de *HighStock*.

```
1  <script src="http://code.highcharts.com/stock/highstock.js"></
       script>  // Cargar el framework
2
3  myChart = new Highcharts.StockChart({
4      chart: {
5          renderTo: container}, //HTML element reference
6      series: [{
7          name: 'My First Series',
8          data: myData // predefined JavaScript array
9      }]
10 });
```

Como podemos ver para crear un gráfico invocamos la función `StockChart()`. Esta función acepta como parámetro un objeto que define la configuración del gráfico. El formato de este objeto puede verse en la documentación del framework[4]. En nuestro caso tan sólo especificaremos dos atributos. La mayoría de los atributos tienen un valor por defecto por lo que no es necesario especificarlos explícitamente.

El primer atributo que fijamos es `chart`. Este atributo es un objeto que a su vez determina aspectos básicos del gráfico. El atributo que especificamos es `renderTo`, el elemento HTLM en el que nuestro gráfico será presentado.

El segundo atributo es `series`. Este es un array de objetos que definen las series de nuestro gráfico. En este ejemplo tenemos tan sólo un objeto, por lo que nuestro gráfico tendrá tan sólo una serie. Definiendo el atributo `name` le damos un nombre a nuestra serie y con el atributo `data` le damos datos a la misma. El atributo `data` tiene que ser un array que contenga los datos. Este array puede tener múltiples formatos que se pueden utilizar según convenga. Para ello se puede consultar la documentación proporcionada[4].

En este punto volveremos al `renderTo` a fin de explicar la integración entre *HighStock* y *Sencha ExtJs*. En secciones anteriores explicamos que la interfaz de usuario de una aplicación *ExtJs* se constituye mediante componentes. Estos componentes son abstracciones con funcionalidades muy extendidas pero en el fondo son elementos HTML. El atributo `renderTo` puede tomar el valor de una de las etiquetas HTML, a fin de mostrar un gráfico *HighStock* en un componente de una aplicación *ExtJs*. El elemento HTML de un componente es accesible de la siguiente forma.

```
Ext.Component.getEl().dom
```

5.2.1. Agrupación de datos y *Candlestick*

El objetivo de esta sección es explicar el gráfico *Candlestick* o gráfico de velas. Primero explicaremos el motivo por el que utilizamos este tipo de gráfico y seguidamente como interpretarlo.

El problema radica en la gran cantidad de datos que tienen que ser representados. El sistema de adquisición genera una entrada de datos cada minuto, 1440 datos al día o 43200 datos al mes aproximadamente. Estas cantidades de datos son excesivas comparadas con las resoluciones de pantalla actuales. Consideremos una pantalla estándar con 1280 píxeles de ancho, es imposible representar 43200 datos.

Para dar solución a este problema hemos recurrido a agrupar los datos. El fin es crear un número de grupos que no exceda el ancho de píxeles de nuestra pantalla. Para cada grupo tenemos que calcular un valor representativo. Una técnica común es calcular el valor medio de los grupos. Esto tiene el inconveniente de aplanar los Spikes, algo que queremos resaltar.

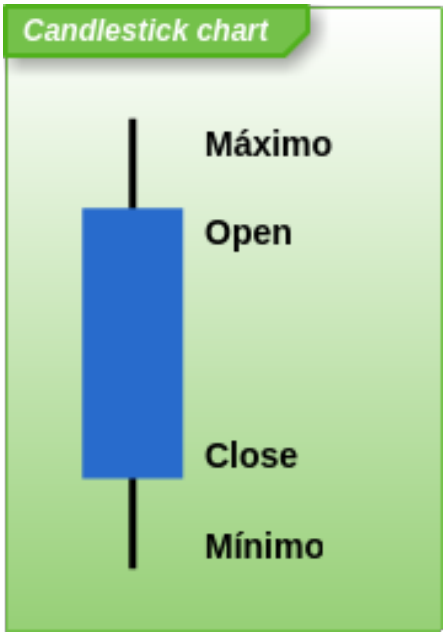

Figura 5.4: Gráfico Candlestick.

Para evitar aplanar los Spike para cada grupo calcularemos valores más representativos que la media. En concreto calcularemos cuatro valores, máximo, mínimo, *open* y *close*. El *open* y *close* son la media más la desviación típica y la media menos la desviación típica respectivamente.

El agrupamiento de datos y cálculo de estos cuatro valores para cada grupo está implementado en el *back-end*. Los servicios RPC que nos permiten acceder a estos datos son `nmdbOriginalGroup` y `nmdbRevisedGroup`. Estos servicios fueron explicados en el capítulo dedicado al *back-end*. En este punto tan sólo nos interesa destacar el argumento `points` que estos dos servicios aceptan. Este argumento especifica el número de conjuntos en los que serán agrupados los datos, que será proporcional al número de píxeles que tenemos disponibles para representar nuestro gráfico.

En análisis técnico bursátil los valores *open* y *close* tienen su significado propio, en este trabajo representan la media más la desviación típica y la media menos la desviación típica respectivamente. Como podemos ver en la figura 5.4 el *open* y *close* son utilizados para construir el cuerpo de las velas. Este representa los datos que se encuentran dentro de la desviación típica. La línea que cruza el cuerpo de la vela representa el valor máximo y mínimo para cada grupo.

Las velas con cuerpos alargados indican grupos en los que la desviación típica es muy grande, por contrario velas con cuerpos contraídos indican grupos con una desviación típica pequeña. Los dos extremos, datos que fluctúan mucho o datos que no fluctúan, pueden ser indicativo de un mal funcionamiento. Los valores máximo o mínimo que se alejan mucho del cuerpo de la vela nos indican la presencia de un Spike en dicho grupo.

5.2.2. Ampliando HighStock

HighStock es construido de forma modular para facilitar la ampliación por parte de los desarrolladores. En este trabajo hemos ampliado el framework con dos funcionalidades adicionales. El propósito de esta sección es describir estas dos funcionalidades, pero primero explicaremos el proceso para ampliar el framework.

La primera consideración que tenemos que tener es el ámbito o *scope*, nombre que utilizaremos a lo largo de este trabajo. Debemos tener cuidado de no contaminar el *scope* global con variables. Para este propósito debemos utilizar una función *self-invoking*. En *JavaScript* esta es una función que se ejecuta inmediatamente y crea su propio *scope*, de esta manera evitamos contaminar el *scope* global. A continuación presentamos un ejemplo de como declarar una de estas funciones.

```
1 (function (H) {
2     var localVar,         // Variable local
3     Series = H.Series;
4     doSomething();
5 }(Highcharts));
```

El framework ofrece la función `wrap` que permite añadir código en una función existente. Podemos añadir dicho código antes o después del código inicial de la función. La función acepta tres argumentos, el objeto padre como primer argumento, el nombre de la función como segundo argumento y la función de sustitución como tercer argumento. La función original se pasa como el primer argumento de la función de sustitución. Es conveniente fijarnos en el ejemplo de código a continuación presentado para entender mejor el uso de la función `wrap`.

```
H.wrap(H.Series.prototype, 'drawGraph', function (proceed) {
// Antes de la función original.
console.log("We are about to draw the graph: ", this.graph);
// Invocar la función original usando todos los argumentos menos
    el primero.
proceed.apply(this, Array.prototype.slice.call(arguments, 1));
// Después de la función original.
console.log("We just finished drawing the graph: ", this.graph);
});
```

A continuación explicaremos las dos ampliaciones que hemos realizado sobre el framework. La primera permite realizar *zoom out* y la segunda permite resaltar las series de un gráfico para poder inspeccionarlas mejor.

Zoom Out

Tal y como explicamos los gráficos creados con el framework son altamente interactivos. Entre las diferentes funcionalidades están las que permiten navegar por el conjunto de datos. Con navegar nos referimos a mostrar diferentes fragmentos de los datos. Un ejemplo de estas funcionalidades es el *zoom in*, que permite ampliar una sección de los datos. Haciendo *click* y arrastrando, sin importar la dirrección, podemos seleccionar un intervalo para realizar el *zoom in*. Esto es muy útil para inspeccionar los datos, el problema esta en que el framework no ofrece la posibilidad de realizar un *zoom out*.

En este trabajo hemos extendido el framework para incorporar esta funcionalidad. Para este propósito utilizamos la dirección de arrastre al realizar una selección. Un arrastre de izquierda a derecha realiza un *zoom in*, mientras que un arrastre de derecha a izquierda realiza un *zoom out*.

Para poder registrar la dirección de arrastre hemos hecho uso de dos eventos, `mousedown` y `mouseup`. En cada evento registramos la posición de este y usamos la diferencia entre las posiciones para determinar la dirección de arrastre. A continuación presentamos el código. Es la variable `selectDirection` la que indica la dirección de arrastre.

```
H.addEvent(container, 'mousedown', function (e) {
    selectFrom = chart.pointer.normalize(e).chartX;
});

H.addEvent(container, 'mouseup', function (e) {
    selectTo = chart.pointer.normalize(e).chartX;
    selectDirection = selectTo - selectFrom;
});
```

El evento que se dispara al realizar una selección es `selection`, por lo que tenemos que registrar un manejador para este. La primera acción es comprobar el estado del *selectDirection*, la variable que indica la dirección de arrastre. En el caso de que la variable tenga un valor positivo no tomamos acción, dejamos que se realice la accion por defecto que es *zoom in*. Si por contrario la variable tiene un valor negativo tenemos que prevenir la acción por defecto. Seguidamente calculamos los valores que definen el nuevo intervalo temporal y, para terminar aplicamos el nuevo intervalo al gráfico. A continuación se puede ver el código que implementa la funcionalidad previamente descrita.

```
H.addEvent(chart, 'selection', function (e) {
    if (selectDirection < 0) {
        e.preventDefault();
        //Calcular newMin y newMax
        xAxis.setExtremes(newMin, newMax);
    }
});
```

Resaltar series

En la mayoría de casos los monitores de neutrones se componen por 18 tubos contadores. Los gráficos que representan la información de los tubos por separado se vuelven confusos debido al gran número de series. Este problema crea la necesidad de tener un mecanismo que permita resaltar una serie del gráfico para poder inspeccionarla mejor. Para crear dicho mecanismo utilizaremos la leyenda del gráfico.

HighStock permite añadir una leyenda a los gráficos. Esta leyenda consiste en un elemento gráfico que facilita el identificar e interpretar las diferentes series del gráfico. La leyenda, al igual que los gráficos, es altamente interactiva. Haciendo *click* sobre los elementos de una leyenda podemos mostrar u ocultar la serie correspondiente. En este trabajo preservamos esta funcionalidad y añadimos la funcionalidad de resaltar una serie. Para este propósito utilizamos los eventos `mouseover` y `mouseout`, de forma que la serie se resalta al posicionar el puntero sobre el elemento correspondiente de la leyenda.

El proceso de resaltar una serie para poder visualizarla mejor consiste en bajar la opacidad de todas las demás series, de esta manera la serie que queremos resaltar es más visible. En la figura 5.5 podemos ver el resultado que obtenemos. A la izquierda podemos ver el gráfico antes de resaltar una serie, vemos que las mediciones de los 18 canales se solapan y es difícil examinarlas. A la derecha podemos ver el canal 13, *ch13*, resaltado.

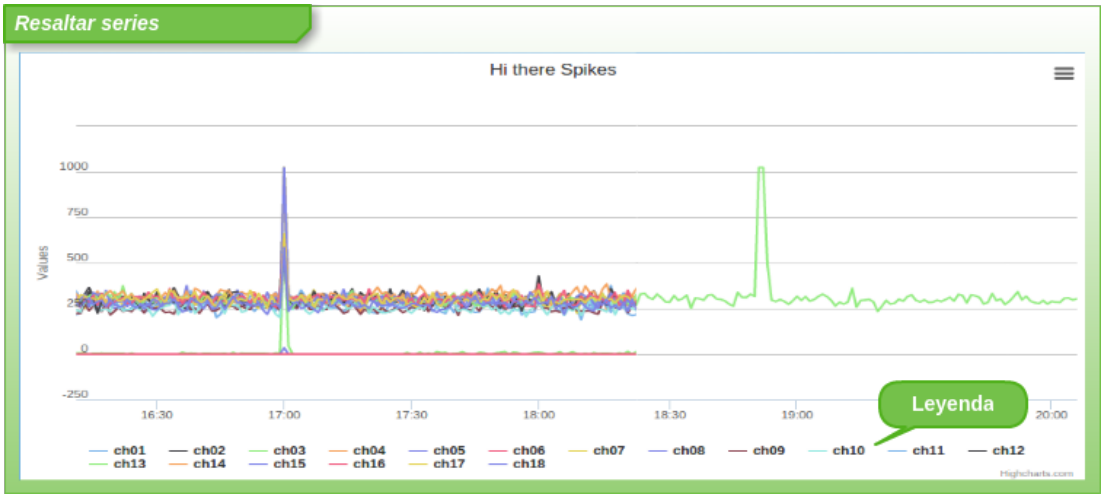

Figura 5.5: Ampliando *HighStock*. Resaltar series.

5.3. Modelo

El *modelo* es el encargado de manejar los datos de una aplicación. En el caso del *front-end* los datos de la aplicación deben ser servidos por el *back-end*. El *modelo* es el encargado de realizar la comunicación con el *back-end*.

Tal y como especificamos en el capítulo anterior el protocolo para la comunicación entre los dos módulos es *HTTP*. Nuestro *front-end* es el que empieza la comunicación enviando un mensaje de petición y el *back-end* responde a esa peticion con un mensaje de respuesta. El *modelo* es el encargado de enviar los mensajes de petición y después interpretar los mensajes de respuesta.

Para dotar al *modelo* de la funcionalidad necesaria utilizamos las facilidades que *ExtJs* ofrece, concretamente utilizamos el singleton `Ext.Ajax`. *Ajax*[38] es una técnica de desarrollo Web donde cliente y servidor mantienen una comunicación asíncrona en segundo plano. `Ext.Ajax` es un singleton de la clase `Ext.data.Connection`, clase que encapsula la lógica necesaria para realizar una comunicación *Ajax*.

Concretamente hacemos uso de la función `Ext.data.Connection.request`. Esta función envía una petición *HTTP* a un servidor remoto. La función acepta únicamente un parámetro que es un objeto cuyas propiedades definen el comportamiento de la función. Las propiedades de las que nosotros hacemos uso están detalladas a continuación.

- `url`. La URL a la que enviaremos la petición. En nuestro caso *back-end* y *front-end* estarán albergados en el mismo *host*. Esto nos permite no especificar el *host*, la petición se hará al *host* utilizado para cargar la aplicación. En la URL solamente hay que especificar el servicio deseado y los parámetros que acompañan a este. Seguidamente presentamos un ejemplo del valor que puede tomar el campo `url`.

```
url: '/nmdadb/channel/stats/default/default'
```

- `method`. El método que especifica nuestro mensaje de petición. Los mensajes *HTTP* de petición pueden especificar un método. Si este campo se deja vacío el método utilizado por defecto es `GET`. La mayoría de los servicios ofrecidos por el *back-end* aceptan un método `GET`, pero el servicio `nmdbMarkNull` acepta un método `POST`.

- `success`. La función a ser invocada al completar con éxito la petición. Esta función a su vez acepta como parámetro `response`. Este parámetro contiene los datos del mensaje de respuesta.

- `failure`. La función a ser invocada al completar sin éxito la petición. Esta función también acepta el parámetro `response` que podemos utilizar para identificar la causa del fallo.

- `timeout`. El número de milisegundos en los que el *back-end* debe responder. Si el tiempo expira la solicitud se considera como fallida.

Más allá de `Ext.Ajax` el framework ofrece abstracciones de un nivel superior. La clase `Ext.data.Model` es una representación de un objeto utilizado por nuestra aplicación. Estos modelos son usados por la clase `Ext.data.Store`, que encapsula instancias de estos. Estas abstracciones son muy útiles, pero algo complejas. Por esta razón la mayoría de los datos se manejan usando el `Ext.data.Connection`, mientras que hemos

utilizado el `Ext.data.Store` en casos muy específicos como los datos que necesitan ser mostrados en una tabla. Las tablas en *ExtJs* deben tener asociado un `Ext.data.Store` cuyos datos mostrar. A continuación podemos ver un ejemplo de como declarar un `Ext.data.Model` y un `Ext.data.Store` que enlazaremos a un `Ext.grid.Panel`.

```
1  Ext.define('MyApp.model.MyModel',{
2     extend: 'Ext.data.Model',
3     fields: [    {name: 'myName1', type: 'string'},
4                  {name: 'myName2', type: 'string'}]
5  });
6
7  Ext.define('MyApp.store.MyStore',{
8     extend: 'Ext.data.Store',
9     model: 'MyApp.model.MyModel',
10    proxy: {
11       type: 'ajax',
12       url: '/nmdadb/channel/stats/default/default',
13       reader: {
14          type: 'json'}}
15 });
16 var myStore = Ext.create('MyApp.store.MyStore');
17 myStore.load();
18
19 Ext.define('MyApp.grid.MyGrid',{
20    extend: 'Ext.grid.Panel',
21    title: 'MyCoolGrid',
22    store: myStore,
23    fields: [    {text: 'Name1', dataIndex: 'myName1'},
24                 {text: 'Name2', dataIndex: 'myName2'}]
25 });
26 Ext.create('MyApp.grid.MyGrid', {
27    renderTo: Ext.getBody()
28 });
```

Podemos ver que en la declaración del `Ext.data.Store` especificamos `proxy`. El objeto especificado se utiliza para crear una instancia de la clase `Ext.data.proxy.Proxy`. Esta clase es una abstracción que agrupa diferentes técnicas de comunicación como *Ajax*, *JsonP*, *Rest* y *Direct*.

5.4. Controlador

El Controlador en una aplicación *modelo-vista-controlador*[10] hace de intermediario entre la Vista y el Modelo. Normalmente este responde a los eventos producidos por el usuario. Las respuestas consisten en peticiones al Modelo para cargar datos o bien en comandos dirigidos a la Vista para realizar un cambio en la forma de mostrar esos datos. El Controlador contiene toda la lógica de la aplicación por lo que podemos decir que es el módulo más interesante. El propósito de esta sección es hacer una descripción de este.

Sencha ExtJs ofrece la clase `Ext.app.Controller`, que representa la abstracción de un Controlador. Esta clase contiene la funcionalidad mínima necesaria, está pensada para ser extendida y dotada de funcionalidad por el usuario. A continuación podemos ver un pequeño ejemplo de como declarar un Controlador básico en *Sencha ExtJs*.

```
1 Ext.define('MyApp.controller.Users', {
2     extend: 'Ext.app.Controller',
3     onLaunch: function() {
4         console.log('Podemos empezar..');}
5 });
```

La función `onLaunch()` es una función especial que es invocada después de inicializar la aplicación. La Vista y Modelo ya están inicializados, por lo que es un buen punto para empezar a actuar sobre estos elementos.

Antes de empezar a discutir los Controladores que hemos implementado en este trabajo es conveniente destacar que el framework ofrece una facilidad similar al `Ext.ComponentQuery`, pero para los Controladores. Podemos hacer consultas para acceder a los Controladores desde cualquier punto del código. A continuación podemos ver como hacer uso de esta facilidad.

```
MyApp.app.getController("Nombre_del_Controlador")
```

En nuestra aplicación hemos definido cinco Controladores, que se describirán en las las subsecciones venideras.

5.4.1. `HighStockExtend`

El propósito de este Controlador es inicializar las dos extensiones de *HighStock* que discutimos previamente en este capítulo. Este Controlador se invoca antes de empezar a construir los gráficos de la aplicación para que las extensiones estén listas y sean aplicadas a estos.

5.4.2. `Navigation`

Este Controlador es el encargado de inicializar el `Ext.History`,que es el mecanismo que soluciona el problema de navegación por el historial en aplicaciones *single-page*. El problema y el mecanismo fueron descritos en secciones anteriores de este capítulo.

El Controlador define la función `onLaunch()`, que se invoca una vez creados la Vista y el Modelo de la aplicación. Empieza invocando la función `Ext.History.init()`, a fin de inicializar este módulo. Seguidamente define la función encargada de manejar el evento `change`, que se dispara al cambiar el *hash* de la URL. Esta función soporta tres valores diferentes como *hash* que son `[Spike, SpikeRevised, ChannelStats]`. Estos tres valores representan los tres módulos funcionales entre los que hacemos distinción. Cambiando el *hash* cambiamos el módulo funcional mostrado en pantalla.

Podemos ver que el soporte de navegación por el historial es pobre en funcionalidad, por esta razón proponemos como trabajo futuro extender y ampliar la funcionalidad de este.

También debemos tener en cuenta que al inicializarse la aplicación puede haber un *hash* presente. Una vez especificado el comportamiento ante un evento `change` procedemos a evaluar el *hash* actual. Este puede no estar presente, caso en el que cargamos el módulo funcional `Spike`. En el caso de la presencia de un *hash* actuamos conforme a este.

5.4.3. `Spike`

Este Controlador es directamente relacionado al módulo funcional con el mismo nombre, `Spike`. Este Controlador es algo más complicado que los anteriores, dado

que se compone de múltiples funciones. A continuación procedemos a explicar estas funciones.

`Launch()` Inicializa el módulo funcional `Spike`. Es conveniente destacar que esta función no es `onLauch`, y por lo tanto no es invocada automáticamente al inicio de la aplicación, sino que somos nosotros los que la invocamos. Esta función inicializa el objeto `app`, que debe contener todas las variables usadas en este módulo. La mayoría de las demás funciones usaran este objeto para acceder a estas variables. Después de inicializar este objeto se invoca `loadInitialData()`.

`loadInitialData()` Esta función carga los datos necesarios para construir el módulo. Los datos son cargados realizando una petición *Ajax* al *back-end*. El servicio RPC que invocamos es `nmdbOriginalGroup`. Al cargar los datos con éxito se invocann a su vez las funciones `initCandleChart()` y `initChannelChart()`.

`initCandleChart()` Esta función es la encargada de crear el gráfico *Candlestick* con los datos de la estación. Hablaremos más a fondo sobre este gráfico en las secciones futuras donde discutiremos la herramienta desde un enfoque funcional.

`initChannelChart()` Esta función es la encargada de crear el gráfico que representa los datos de los tubos contadores por separado. Igual que en el caso anterior hablaremos más sobre este gráfico en secciones futuras.

`LineOrOhcl(start, finish, N_points)` Anteriormente en este capítulo discutimos el problema que surge al intentar representar un gran número de datos en un gráfico convencional y que el gráfico *Candlestick* da solución a este problema. Esta función determina si debemos utilizar un gráfico de línea o bien uno *Candlestick*. La cantidad de datos se define por los parámetros `start` y `finish`, mientras que el parámetro `N_points` representa el número de píxeles que tenemos a disposición para representar los datos.

`updateMode(start, finish)` Esta función es invocada cuando ocurre algún cambio en el gráfico *Candlestick* y calcula el modo en que deben ser representados los datos. Esta función hace un uso intensivo de la función `LineOrOhcl()`, pero tiene en cuenta más cosas como la serie actual.

`changeSeries(series)` Esta función cambia la serie que es presentada en el gráfico *Candlestick*. El parámetro `series` es un entero que puede tomar los siguientes valores `[1, 2, 3]`, donde estos representan *uncorrected*, *corrected for pressure* y *corrected for efficiency*.

`updateSeries()` Esta función simplemente refresca los datos de las series del gráfico *CandleStick*, y se invoca cada vez que ocurre algún cambio en este para refrescar los datos.

`updateCandleData(start, finish)` Esta función carga los datos para el gráfico *Candlestick* en función de la petición realizada por el usuario. Para cargar los datos la función realiza una petición *Ajax* al *back-end*. La función invoca el servicio RPC `nmdbOriginalGroup` o `nmdbOriginalRaw` en función del valor devuelto por la función `LineOrOhcl()`. El intervalo de datos que será solicitado se define por los parámetros `start` y `finish`.

`updateChannelData()` Esta función carga los datos para el gráfico que representa los datos de los tubos contadores por separado. Estos datos se descargan desde el *back-end* mediante una petición *Ajax* que invoca el servicio RPC `nmdadbRawData`.

`searchInterval(start, finish)` Esta función permite hacer una búsqueda por intervalos temporales. El intervalo es definido por los parámetros `start` y `finish`.

`getTimestamp(str)` Esta función acepta una cadena de texto que debe seguir un formato determinado y devuelve un objeto de la clase `Date`.

`showCandle()` Esta función muestra el gráfico `Candlestick` creado por este Controlador. Junto al gráfico también se asegura de mostrar los controles vinculados a este. Si es la primera vez que intentamos mostrar este gráfico el módulo entero no estará creado, por lo que es invocada la función `Launch()`.

`showChannel()` Esta función muestra el gráfico de los tubos contadores por separado. Junto al gráfico también se asegura de mostrar los controles vinculados a este. Es invocada la función `updateChannelData()` para asegurar que son mostrados los datos correctos.

5.4.4. `SpikeRevised`

Este Controlador es muy similar al anteriormente descrito, `Spike`. Las diferencias como indica el nombre radican en que este utiliza los datos revisados, no los originales. Además de utilizar un conjunto de datos distinto este Controlador también ofrece funcionalidad extendida, permite marcar datos como inválidos. Los datos marcados como inválidos serán considerados nulos en el conjunto de datos revisados.

A continuación procederemos a explicar las funciones que este Controlador define.

`Launch()` Inicializa el módulo funcional `SpikeRevised`. Esta función inicializa el objeto `app` que debe contener todas las variables asociadas a este módulo. La estructura de este objeto es idéntica al objeto creado en la función `Launch()` del Controlador `Spike` a excepción de una serie de atributos extra.

Seguidamente se importan todas las funciones que podemos reutilizar del Controlador `Spike`, de esta forma se evita duplicar código idéntico. Las funciones importadas son las siguientes:

```
initCandleChart, initChannelChart, changeSeries, searchInterval,
updateChannelData, LineOrOhcl, updateMode, updateSeries, getTimestamp
```

Finalmente se llama a la función `loadInitialData()`.

`loadInitialData()` Esta función es muy parecida a la función con el mismo nombre del Controlador `Spike`. La función carga los datos necesarios para construir el módulo. La diferencia radica en el servicio RCP utilizado. Esta función utiliza el servicio `nmdbRevisedGroup`, tal y como explicamos este módulo utiliza los datos revisados.

Al cargar los datos con éxito son invocadas las funciones `initCandleChart` y `initChannelChart` que crean los dos gráficos, estas son función importadas del Controlador `Spike`. Finalmente se invoca la función `extendChandleChart` que se explicará a continuación.

`extendCandleChart()` Volvemos a recordar que este módulo es muy similar al `Spike`, pero ofrece alguna funcionalidad extra. Esta función es la encargada de extender la funcionalidad del gráfico *Candlestick*. La función define el comportamiento de las series del gráfico ante el evento `click`. Este comportamiento consiste en marcar el dato con un flag y añadirlo a una tabla. Posteriormente los datos en la tabla podrán ser clasificados como inválidos.

`updateCandleData(start, finish)` Esta función es muy parecida a la función con el mismo nombre del Controlador `Spike`. La función carga los datos para el gráfico Candlestick en función de la petición realizada por el usuario. La diferencia radica en los servicios RPC utilizados. Esta función utiliza lo servicios `nmdbRevisedGroup` o `nmdbRevisedRaw` en función del valor devuelto por la función `LineOrOhcl()`.El intervalo de datos que es solicitado es definido por los parámetros `start` y `finish`.

`removeFromGrid(rowIndex)` Los datos marcados son guardados en una tabla. Esta función permite eliminar un dato de esta tabla. El parámetro que esta función acepta especifica el índice del dato que será eliminado de la tabla.

`submitGrid()` Esta función permite clasificar como inválidos los datos en la tabla. La función invoca el servicio `nmdbMarkNull` del *back-end* a fin de cumplir con su propósito. Una vez anulados correctamente los datos, la tabla se vacía y el gráfico se refresca con el fin de mostrar los cambios.

`showHideGrid()` Esta función permite mostrar u ocultar la tabla que contiene los datos marcados, de esta manera los datos pueden ser inspeccionados.

`clearGrid()` Esta función permite vaciar la tabla que contiene los datos marcados, de esta manera estos son descartados y no se toma ninguna acción.

`showCandle()` Esta función es muy parecida a la función con el mismo nombre del Controlador `Spike`. La función muestra el gráfico *Candlestick* y todos los controles vinculados a este. La función `Launch()` es invocada en el caso de que el módulo no esté creado.

`showChannel()` Esta función es muy parecida a la función con el mismo nombre del Controlador `Spike`. La función muestra el gráfico de los tubos contadores por separado. Junto al gráfico también se asegura de mostrar los controles vinculados a este. Es invocada la función `updateChannelData()` para asegurar que son mostrados los datos correctos.

5.4.5. `ChannelStats`

Este Controlador está directamente relacionado con el módulo funcional con el mismo nombre, `ChannelStats`. Este Controlador se compone de varias funciones que discutiremos a continuación.

`Launch()` Inicializa el módulo funcional `ChannelStats`. Esta función inicializa el objeto `app`, que debe contener todas las variable usadas en este módulo. Después de iniciar este objeto se invoca la función `loadInitialData()`.

`loadInitialData()` Esta función carga los datos necesarios para construir el módulo. Los datos son cargados realizando una petición *Ajax* al *back-end*. Se invocan dos servicios RPC, `nmdadbChannelStats` y `nmdadbChannelHistogram`. El primero devuelve

una serie de estadísticas descriptivas de los canales. Estos datos se representan en un gráfico y también se muestran en una tabla. El segundo carga los datos necesarios para construir un histograma de la distribución de cuentas para los 18 canales.

Al cargar los datos con éxito se invocan las funciones `initStatsChart()` y `initHistogramChart()` que crean los dos gráficos: el gráfico con las estadísticas y el histograma. Los datos de la tabla son actualizados automáticamente, por lo que no tenemos que tomar ninguna acción.

`initStatsChart()` Esta función es la encargada de crear el gráfico a partir de los datos devueltos por el serviocio RPC `nmdadbChannelStats`.

`initChannelHistogram()` Esta función es la encargada de crear el histograma que representa la distribución de cuentas para los 18 canales de la estación.

`searchInterval()` La función que carga los datos iniciales carga los datos del último mes, esta función permite solicitar otro intervalo temporal. Este intervalo se define por los dos parámetros que esta función acepta, `start` y `finish`. La función invoca los servicios RPC `nmdadbChannelStats` y `nmdadbChannelHistogram` para solicitar los datos al *back-end*.

`showModule()` Esta función muestra en pantalla este módulo funcional. En el caso de que este módulo no esté creado se invoca la función `Launch()`.

5.4.6. Visión general

En la figura 5.6 podemos ver un diagrama que representa los Controladores que acabamos de explicar. En color azul podemos ver las funciones, gris en caso de que sean funciones importadas desde otro Controlador. Las flecha azules que unen las funciones muestras como estas se invocan las unas a las otras. Algunas funciones son invocadas al producirse un cierto evento, estos pueden verse en verde dentro de las funciones. Todos los eventos son producidos por los elementos que forman la Vista de la aplicación, a excepción del evento `Ext.History.change` que se dispara al realizar un cambio en la pila del historial de navegación.

5.5. Vista

El componente de Vista en una aplicación *modelo-vista-controlador*[10] es el responsable de presentar la información de la aplicación junto a todos los componentes que componen la interfaz de usuario. El propósito de esta sección es describir los aspectos básicos de este componente.

La Vista en una aplicación *ExtJs* se compone de una jerarquía de *componentes*. En nuestro caso en lo más alto de esta jerarquía tenemos un `ViewPort` al que hemos asignado el nombre `MyViewPort`. Este es un *componente* que se ajusta al área de aplicación disponible. El layout que determina como se posicionan los hijos de este *componente* es `border`. Los *componentes* hijos deben especificar el atributo `region` que determina su posicionamiento.

El primer hijo con `region:north` es un `Ext.container.Container`, que especifica un *contenedor* base. Actualmente este *contenedor* no tiene ninguna funcionalidad. En un futuro se plantea visualizar en él mensajes de alerta.

El segundo hijo del `MyViewPort` con `region:west` es también una instancia de la clase `Ext.container.Container`. El `itemId` que define es `Controls`. Este *contenedor*

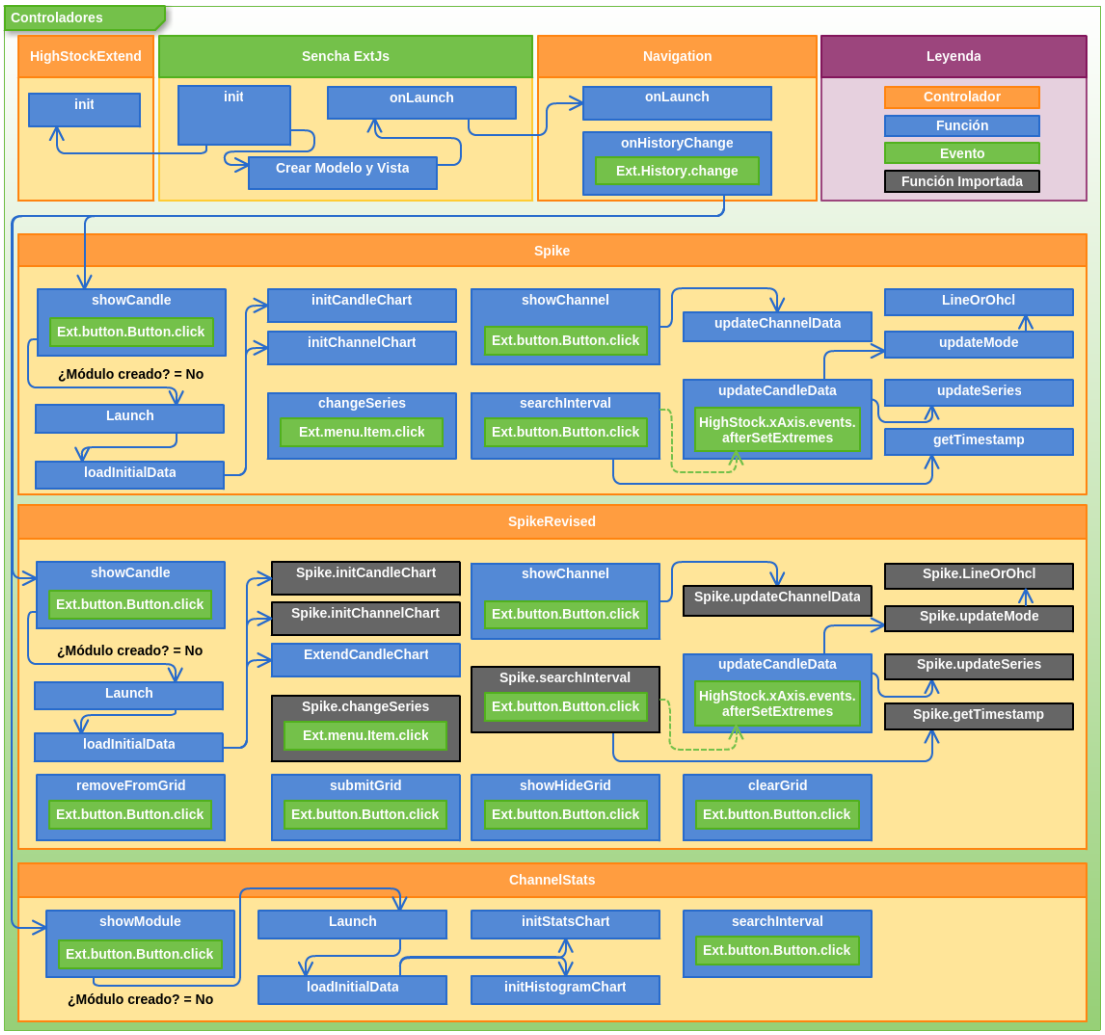

Figura 5.6: Visión general de los Controladores.

tiene tres hijos que son instancias de la misma clase. Estos tres hijos contienen los controles vinculados a los tres módulos funcionales. Los `itemId` que estos tres hijos especifican son:

```
[SpikeControls, SpikeRevisedControls, ChannelStatsControls]
```

`Controls` usa el layout `accordion`, donde solamente uno de los hijos puede estar expandido y los demás están colapsados, de esta manera no pueden ser visibles lo controles de dos módulos funcionales a la vez. Cada vez que se expanden los controles de un módulo funcional se dispara el evento `Ext.panel.Panel.beforeexpand`. El handler asociado a este evento invoca la función apropiada de los Controladores para mostrar el módulo funcional al que pertenecen los controles que acaban de ser expandidos.

La tercer hijo del `MyViewPort` con `region:center` es también una instancia de la clase `Ext.container.Container`. El `itemId` que identifica a este componente es `NavigationPanel`. El layout utilizado es `card` donde los hijos se solapan unos a otros y tan sólo uno puede ser visible a la vez. Los *componentes* hijos contienen todos los gráficos y tablas que nuestra aplicación ofrece. Estos se muestran y ocultan en función del estado de los controles.

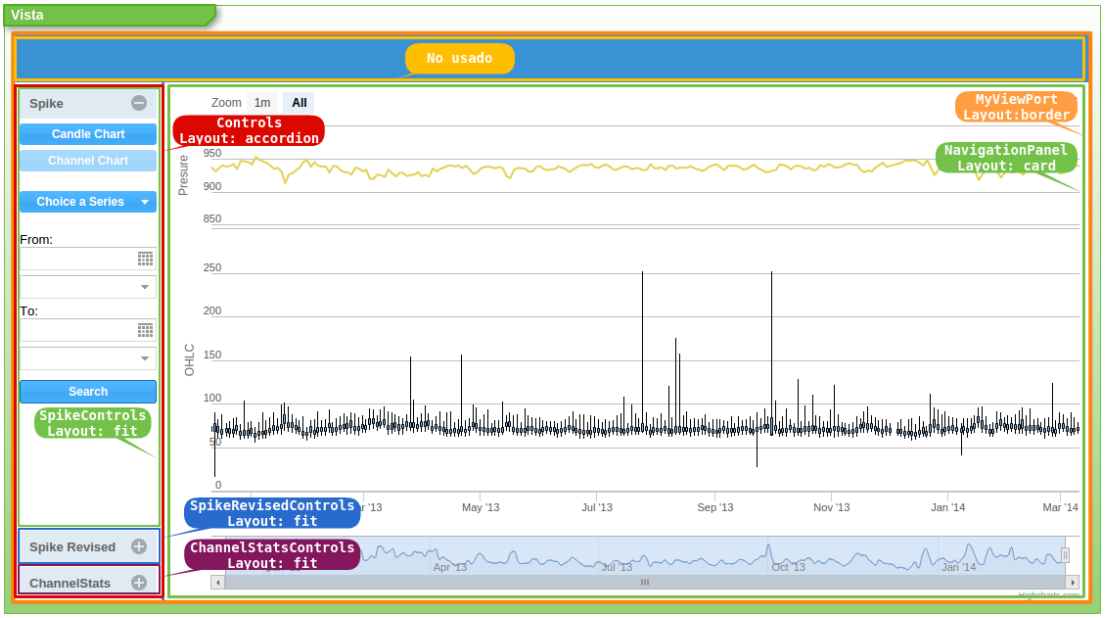

Figura 5.7: *Front-end*. Vista.

En la figura 5.7 podemos ver como se organizan los componentes de la Vista que acabamos de explicar.

5.6. Descripción funcional

El propósito de esta sección es hacer una descripción funcional de la herramienta. Diferenciamos entre tres módulos funcionales, que son `Spike`, `SpikeRevised` y `ChannelStats`.

5.6.1. `Spike`

El propósito de este módulo es ofrecer un gráfico interactivo que permite localizar los *spikes* de forma fácil. Recordamos que estos son datos anormalmente grandes o pequeños. La figura 5.8 presenta el módulo funcional `SpikeRevised`. Dado que los dos módulos son muy similares podemos fijarnos en esta figura.

Inicialmente el módulo tiene que representar todos los datos de la estación, razón por la que se utiliza el gráfico *Candlestick*. En este gráfico los valores máximo y mínimo para cada grupo son fáciles de distinguir. Eventualmente cuando se solicita un intervalo temporal para el que no es necesario agrupar los datos, el gráfico automáticamente cambia a un gráfico de línea. Por contrario si se vuelve a solicitar un intervalo que requiere agrupar los datos, el gráfico cambia a *Candlestick*.

Para solicitar un intervalo diferente de datos tenemos varias formas. La primera es el *zoom* interactivo que podemos realizar haciendo *click* y arrastrando sobre un intervalo del gráfico. Dependiendo de la dirección de arrastre este será un *zoom in* u *out*. La segunda alternativa es utilizar los campos de entrada disponibles en los controles para realizar una búsqueda por intervalo. En estos campos podemos especificar el inicio y fin del intervalo y la búsqueda será realizada al accionar el botón `Search`. La tercera opción es utilizar el navegador que podemos encontrar en la parte inferior del gráfico.

Finalmente podemos utilizar los botones en la parte superior izquierda del gráfico, `1m` y `All`, que permiten fijar un intervalo de un mes o volver a mostrar todos los datos.

Inicialmente se muestran los datos *sin corregir*, utilizando el botón `Choise a Series` podemos mostrar los datos *corregidos por presión* o los *corregidos por eficiencia*. En la parte superior del gráfico se representan los valores de la presión atmosférica, que permiten relacionar los datos respecto a esta magnitud.

Cuando no es necesario agrupar los datos el botón `Channel Chart` es habilitado. Este botón muestra un gráfico secundario con los valores de los tubos contadores por separado. El intervalo de datos es marcado por el intervalo del gráfico principal. Este gráfico permite observar el comportamiento de los diferentes tubos. El botón `Candle Chart` permite volver al gráfico principal.

5.6.2. `SpikeRevised`

Este módulo incorpora toda la funcionalidad del anterior, la diferencia radica en que este utiliza un conjunto de datos diferente y además ofrece alguna funcionalidad extra. `Spike` utiliza los datos originales, mientras que este módulo utiliza los datos revisados. La funcionalidad extra de este módulo es la que permite añadir datos al conjunto de datos revisados.

Para satisfacer dicha funcionalidad este módulo ofrece una tabla. Esta tabla se muestra en una ventana separada que puede ser mostrada u ocultada utilizando el botón `Selected Points`. Los datos de esta tabla pueden ser enviados al conjunto de datos revisados utilizando el botón `Submit`, que está presente en la misma ventana. Los datos añadidos a este conjunto serán considerados como nulos en el futuro. Este botón además de enviar los datos, actualiza el gráfico para reflejar los cambios que acaban de producirse.

Cuando los datos son representados en modo línea podemos hacer *click* sobre un dato para añadir este a la tabla anteriormente descrita. En el gráfico se añade un *flag*, la letra que aparece en dicho *flag* se corresponde con el campo `Label` de la tabla.

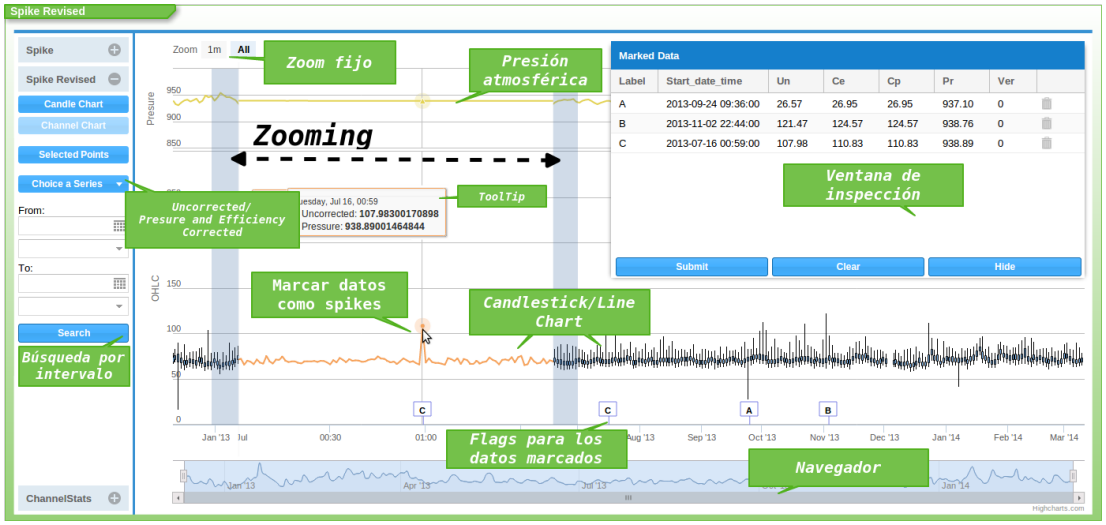

Figura 5.8: Descripción funcional. SpikeRevised

5.6.3. `ChannelStats`

El propósito de este módulo funcional es ofrecer información de los diferentes canales por separado, de esta manera seremos capaces de identificar el mal funcionamiento en canales aislados. El módulo ofrece una tabla en la que podemos inspeccionar la media, desviación típica, máximo, mínimo y último valor para cada canal. Estos datos también son representados en un gráfico *Candlestick*.

Junto a estos datos el módulo ofrece un histograma con la distribución de cuentas para los diferentes canales. Dicho histograma puede consultarse en una pestaña separada.

Inicialmente los datos y el histograma son calculados para el último mes. En los controles podemos encontrar los campos de entrada que permiten solicitar un intervalo de tiempo determinado. La búsqueda por intervalo está limitada a un mes dado a que las consultas necesarias para obtener estos datos no son muy efficientes, estas agrupan los datos por campos que no están indexados. Más sobre este problema podemos encontrar en el capítulo dedicado al *back-end*.

Capítulo 6

Conclusiones y trabajos futuros

El propósito de este capítulo es exponer las experiencias obtenidas a la largo del trabajo y la visión futura que tenemos de este.

6.1. Conclusiones

Las experiencias obtenidas del trabajo son generalmente buenas, el autor considera haber cumplido todos los objetivos inicialmente propuestos y en muchos casos haberlos sobrepasado.

Durante el desarrollo del software de adquisición se han adquirido y reforzado multitud de conocimientos. Las experiencias obtenidas del trabajo conjunto con el equipo de CaLMa han sido muy positivas. El diseño del software ha sido muy condicionado por el hardware ya existente; en sistemas empotrados estos dos están muy enlazados. El autor no considera esto como una desventaja sino como una ventaja. Raras veces en proyectos reales uno tiene libertad total, la mayoría de veces uno se ve forzado a colaborar y adaptarse a las condiciones que le son impuestas. Además el colaborar con personas conlleva el aprender muchas cosas nuevas. Veamos el impacto que han tenido algunas de las decisiones que hemos tomado.

La *BeagleBone Black* ha sido una de las cosas que nos han sido impuestas, sin embargo hemos tenido la libertad de elegir la distribución Linux. Entre *Angstrom* y *Debian*, las dos distribuciones oficiales que la placa soporta, nos quedamos con la primera. A pesar de haber cumplido todas nuestras necesidades esta no es una distribución muy popular y pocas personas están acostumbradas a trabajar con ella. Encontrar información o ayuda también ha sido un desafío debido a la pequeña comunidad de usuarios. Con respeto a esta elección podemos decir que no estamos arrepentidos, sin embargo no es la mejor elección que pudimos tomar.

El desarrollo del software de adquisición en *Python* es una elección que el autor considera afortunada. El gran número de bibliotecas, la gran comunidad existente y todas las propiedades inherentes del lenguaje han permitido avanzar rápidamente en el proceso de implementación. El código generado es claro, corto e intuitivo. La librería de *AdaFruit*[26] nos ha permitido operar sobre el hardware de la *BeagleBone Black* de forma fácil e intuitiva.

Otra elección que consideramos afortunada es utilizar *Sqlite3*[9] para la gestión de la base de datos local. Este sistema de gestión de bases de datos relacionales es muy ligero cosa que lo hace muy adecuado para este trabajo.

La realización de la herramienta Web también ha sido una experiencia agradable en la que hemos adquirido multitud de conocimientos nuevos. A diferencia del software

de adquisición, en la realización de esta hemos tenido más libertad a la hora del diseño. Veamos el impacto que han tenido algunas de las decisiones que hemos tomado. Obviaremos alguna elecciones como la arquitectura *Modelo Vista Controlador* porque son muy triviales y existe gran cantidad de literatura disponible.

Utilizar *Zendframework* y *Apigility* para la realización del *back-end* ha sido una elección buena, sin embargo el autor no esta completamente satisfecho. La combinación entre framework y herramienta han facilitado y acelerado el proceso de implementación. Sin embargo los primeros pasos han sido un poco confusos. El autor considera que este es un framework muy completo pero también muy complicado para principiantes.

La realización del *front-end* en *Sencha ExtJs* es una elección que el autor considera afortunada. El framework es muy completo y en muchos casos implementa todo lo que una aplicación Web pueda necesitar. Trabajar con este ha sido muy fácil e intuitivo, incluso para un principiante.

HighStock es una elección con la que estamos muy satisfechos. La funcionalidad ofrecida ha cubierto todos nuestros requisitos. Además trabajar con el framework ha sido un placer, este es muy intuitivo y podemos encontrar numerosos ejemplos de uso.

A continuación procederemos a detallar los resultados conseguidos en este trabajo, empezando por el software de adquisición. Conjuntamente al sistema de adquisición actual e intentando interferir lo menos posible hemos desplegado el nuevo sistema de adquisición. El objetivo es identificar y eliminar posibles mal funcionamientos que puedan presentarse. El nuevo sistema de adquisición lleva operando de forma parcial desde el 25 de febrero de 2015. Utilizamos la palabra parcial porque hay algunas funcionalidades que no explotamos, siendo la primera de estas es el barómetro (un barómetro como el *BM35* es muy caro, además el proceso de calibración y certificación puede ser aún más costoso). Al no tener barómetro no podemos generar los datos corregidos por presión. Actualmente estamos trabajando en alguna solución que permita a dos o más sistemas de adquisición compartir un barómetro. La segunda funcionalidad que no explotamos son los anchos de pulso. Como explicamos en el capítulo de introducción el nuevo sistema esta diseñado para hacer uso de un circuito de adaptación que genera pulsos cuyo ancho es proporcional a la energía de la señal. Estos circuitos no están listos, en su lugar seguimos utilizando los viejos que generan pulsos cuyo ancho es fijo. Debido a esto los histogramas de ancho de pulso carecen de significado.

En la figura 6.1 podemos ver una captura de la herramienta Web. Este gráfico ha sido generado con los datos registrados por el sistema de adquisición. En relación a la herramienta Web podemos decir poco debido a que no hemos realizado un uso real de esta. Recordamos que en este trabajo tan sólo nos centramos en el proceso de desarrollo. Sin embargo basándose en las opiniones recibidas[17], podemos decir que la herramienta tiene un futuro prometedor.

6.2. Trabajos futuros

A pesar de estar satisfecho con los resultados conseguidos en este proyecto, este podría desarrollarse más.

Tanto el software de adquisición como la herramienta Web han sido desarrollados en el entrono de CaLMa, siendo condicionados por las características propias de la estación. Un ejemplo es el barómetro *BM35* utilizado en CaLMa, en otras estaciones son utilizados otros barómetros. Proponemos como trabajo futuro extender y adaptar tanto el sistema de adquisición como la herramienta Web para otras estaciones.

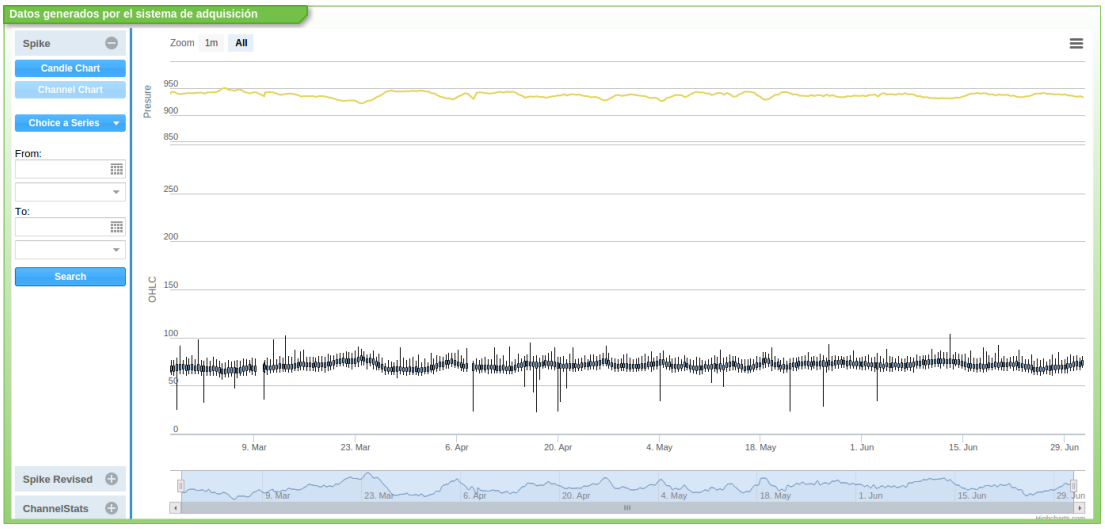

Figura 6.1: Datos generados por el sistema de adquisición.

Volvemos a hacer hincapié, al ser este un sistema empotrado el hardware también podría sufrir modificaciones. Este proceso implicaría implantar y mantener el sistema de adquisición en estas estaciones.

6.2.1. Software de adquisición

A continuación se detallan una serie de mejoras y cambios que el software de adquisición podría experimentar a fin de mejorar.

debian Para la realización del software de adquisición hemos utilizado *Angstrom*, la distribución que viene por defecto con la *BeagleBone Black*. Esta es una distribución muy ligera y adecuada, pero la comunidad de usuarios es muy pequeña. Proponemos como trabajo futuro cambiar la *Angstrom* a alguna distribución derivada de *Debian*. Actualmente existen distribuciones basadas en *Debian* que son compatibles con las placas, gran parte de los usuarios instalan estas distribuciones en sus placas. El fabricante también ha anunciado que en un futuro las placas serán distribuidas con *Debian* por defecto. El inconveniente que este cambio presenta, razón por la que no ha sido realizado en este trabajo, es que el software existente tendrá que sufrir algunos cambios a fin de adaptarse al nuevo entorno.

Actualmente el núcleo Linux y todo el sistema de archivos se cargan desde la memoria interna de la placa. Es posible albergar la nueva distribución en la tarjeta microSD. El contenido de la tarjeta uSD es fácilmente duplicable, podemos exportar este a un archivo de imagen que contendría una copia exacta. Seguidamente podemos grabar este archivo de imagen en una nueva uSD. De esta manera todo el proceso descrito en el apéndice A queda obsoleto, para desplegar el software de adquisición tan sólo tendríamos que grabar un archivo de imagen en una tarjeta uSD e introducirla en la ranura de la *BeagleBone Black*.

6.2.2. Herramienta Web

A continuación se detallan una serie de mejoras y cambios que la herramienta Web podría experimentar a fin de mejorar.

Extender el soporte de navegación por el historial En el capítulo dedicado al *front-end* comentamos que el soporte de navegación por el historial es algo pobre. Proponemos como trabajo futuro extender esta funcionalidad. Actualmente tan sólo son registrados los cambios entre módulos funcionales, es deseable que sean registrados más cambios como los intervalos temporales presentados en los gráficos. De esta manera utilizando los botones de navegación podríamos volver al estado anterior del gráfico.

Anchos de pulso Tal y como explicamos anteriormente en este trabajo el nuevo sistema de adquisición registra los anchos de los pulsos, anchos que son proporcionales a la energía de la señal original. Con estos datos se construyen histogramas con una resolución de 10 minutos que se guardan en la base de datos en forma de cadenas Json. Proponemos como trabajo futuro extender la herramienta con un módulo funcional que permita visualizar estos datos. Para este propósito debemos encontrar la manera más efectiva de presentar los datos. Al igual que el resto de la herramienta este módulo debe ser intuitivo, altamente interactivo y además debe permitirnos especificar diferentes intervalos temporales, ocultar/mostrar series y mucho más.

Configuración del sistema de adquisición Proponemos como trabajo futuro extender la herramienta con un módulo funcional que permita cambiar la configuración del sistema de adquisición en tiempo real. Recordemos del capítulo 3 el archivo de configuración en el que definimos una serie de variables que configuran el software de adquisición. El módulo funcional que proponemos debería permitirnos cambiar el valor de estas variables sin interrumpir el proceso de adquisición. El desarrollo de este módulo tendrá que ser acompañado por el avance del software de adquisición, este debe ofrecer una interfaz que permita esta configuración remota.

Alarmas y Notificación Proponemos como trabajo futuro realizar un módulo funcional que genere alarmas y notificaciones. El módulo funcional debe ser capaz de detectar mal funcionamientos y tomar las acciones oportunas para informar al equipo de la estación. No generar datos o generar datos anormales son algunos ejemplos de mal funcionamiento.

Autenticación y autorización Actualmente la herramienta Web no ofrece ningún servicio de acción, salvo el que nos permite eliminar Spikes. Podemos ver que planeamos realizar más de estos servicios de acción y esto puede ser un problema. Si queremos hacer que la herramienta sea accesible desde Internet debemos implementar algún mecanismo de autenticación y autorización. Los servicios de consulta serian accesible por todos los usuarios, sin embargo los servicios de acción serian accesible tan sólo para los usuarios autorizados.

Implantación y Mantenimiento En el capítulo de introducción especificamos los objetivos de este trabajo, y entre estos objetivos tan sólo contemplamos el desarrollo de la herramienta. Proponemos como trabajo futuro el implantar y mantener el software que compone la herramienta Web.

Apéndice A

Despliegue Software de adquisición

El propósito de este punto es detallar el proceso que debe seguirse para desplegar el software de adquisición. Siendo este un software para un sistema empotrado el estado del hardware tiene una gran influencia, en este punto suponemos que ya está configurado correctamente. También suponemos que sobre la *BeagleBone Black* tenemos una distribución *Angstrom* recién instalada.

A.1. Instalación de software

El primer paso es instalar el software y bibliotecas que son necesarias. Antes de nada debemos actualizar la lista de paquetes disponibles.

```
$ opkg update
```

El primer paquete que debemos instalar es `ntp`, paquete que implementa el protocolo *NTP*. Este paquete permite sincronizar la fecha y hora del sistema. Una vez instalado este paquete debemos actualizar la fecha y hora del sistema, esto evitará posibles problemas.

```
$ opkg install ntp
$ ntpdate -b -s -u pool.ntp.org
```

Nuestro software es escrito en *Python*, debemos asegurarnos que la versión `2.7.x` esta instalada. En caso de no estar presente debemos instalarla.

```
$ python --version
$ opkg install python
```

Seguidamente debemos instalar `pip`, herramienta que facilita el proceso de instalación de bibliotecas *Python*. Utilizando `pip` debemos instalar las bibliotecas necesarias. El uso que hemos hecho de estas bibliotecas está descrito en el capítulo dedicado al software de adquisición.

```
$ opkg install python-pip python-setuptools python-smbus
$ pip install pyserial
$ pip install db-sqlite3
$ pip install MySQL_python
$ pip install Adafruit_BBIO
```

Finalmente debemos instalar *Git*, si este no está presente.

```
$ opkg install git
```

A.2. Clonar repositorio

El software de adquisición ha sido desarrollado utilizando *Git* para llevar un control de versiones. Hemos utilizado *GitHub* para mantener un repositorio remoto. La manera más fácil de acceder al software de adquisición es clonar el repositorio remoto. Antes de clonar el repositorio tenemos que crear un árbol de carpetas similar a este.

```
--/server
   --/server/nmda
   --/server/data
   --/server/logs
```

A continuación procedemos a clonar el repositorio remoto. Una vez clonado el repositorio podemos utilizar el comando `git pull` para actualizar el software. Para utilizar los dos comandos que a continuación presentamos tenemos que estar en la carpeta `/server/nmda`.

```
$ git clone https://github.com/opobla/nmpw.git .
$ git pull
```

En el capítulo dedicado al software de adquisición se explica como este lee una serie de variables desde un archivo de configuración, este archivo debe seguir un formato que también se explica en ese mismo capítulo. Este archivo debe tener el nombre `/server/nmda/.NMDA.conf`. Podemos crear el archivo desde cero o podemos utilizar el archivo de ejemplo que hemos clonado desde el repositorio remoto. Este archivo de ejemplo especifica una configuración mínima que permite correr el software.

```
$ cp /server/nmda/NMDA.conf.exmaple /server/nmda/.NMDA.conf
```

A.3. *System Services*

En el capítulo dedicado al software de adquisición se explica como el sistema de adquisición debe inicializarse automáticamente ante la presencia de corriente. La *BeagleBone Black* por defecto está configurada para arrancar automáticamente. Nos queda configurar el sistema operativo para arrancar nuestro software de forma automática. Para este propósito vamos a utilizar las *System Services*.

Los servicios de sistema se crean mediante archivos con extensión `.service` que deben ser guardados en el directorio `/lib/systemd/system`. A continuación listamos los archivos que definen los servicios que necesitamos.

```
1  # /lib/systemd/system/ntpdate.service
2  [Unit]
3  Description=Network Time Service (one-shot ntpdate mode)
4  Before=ntpd.service
5
6  [Service]
7  Type=oneshot
8  ExecStart=/usr/bin/ntpd -q -g -x
9  RemainAfterExit=yes
10
11 [Install]
12 WantedBy=multi-user.target
```

```
# /lib/systemd/system/ntpd.service
[Unit]
Description=Network Time Service
After=network.target

[Service]
Type=forking
PIDFile=/run/ntpd.pid
ExecStart=/usr/bin/ntpd -p /run/ntpd.pid

[Install]
WantedBy=multi-user.target
```

```
# /lib/systemd/system/nmda.service
[Unit]
Description=Neutron Monitor Data Acquisition Service
After=ntpdate.service

[Service]
ExecStart=/usr/bin/python /server/nmda/NMDA.py

[Install]
WantedBy=multi-user.target
```

```
# /lib/systemd/system/myWatchDog.service
[Unit]
Description=WatchDog

[Service]
ExecStart=/usr/bin/python /server/nmda/WatchDog.py

[Install]
WantedBy=multi-user.target
```

Para habilitar los servicios de sistema que acabamos de declarar tenemos que usar el comando `systemctl`.

```
$ systemctl enable ntpdate.service
$ systemctl enable ntpd.service
$ systemctl enable nmda
$ systemctl enable myWatchDog
```

Vemos que dos servicios utilizan `ntpd`. Para asegurar el correcto funcionamiento de este programa tenemos que editar el archivo de configuración tal y como exponemos a continuación.

```
# /etc/ntp.conf
driftfile /etc/ntp.drift
server pool.ntp.org
restrict default
```

A.4. Tarjeta microSD

Tal y como explicamos en el capítulo dedicado al entorno hardware la *BeagleBone Black* dispone de una memoria integrada (*eMMC*) de 4GB. En esta memoria tenemos el sistema operativo y todo el software necesario para el sistema de adquisición. Esto hace que el porcentaje de ocupación de esta memoria sea muy alto. Es conveniente hacer uso de la ranura *microSD* (*uSD*) para extender la capacidad de almacenamiento. A continuación se detallan los pasos a seguir.

El primer paso es arrancar la *BeagleBone Black* sin la tarjeta *uSD*. Después debemos hacer uso del comando `fdisk`, programa que permite manipular las particiones de los dispositivos de almacenamiento.

```
# Listar las particiones de los dispositivos presentes.
$ fdisk -l
Disk /dev/mmcbk0: 3867 MB, 3867148288 bytes #eMMC

        Device Boot Start      End   Blocks Id System
/dev/mmcblk0p1    *  2048   198655    98304  e W95 FAT16 (LBA)
/dev/mmcblk0p2     198656 7553023 3677184 83 Linux
```

Seguidamente debemos insertar la tarjeta *uSD* y volver a usar el mismo comando.

```
# Listar las particiones de los dispositivos presentes.
$ fdisk -l
Disk /dev/mmcbk0: 3867 MB, 3867148288 bytes #eMMC
        Device Boot Start      End   Blocks Id System
/dev/mmcblk0p1    *  2048   198655    98304  e W95 FAT16 (LBA)
/dev/mmcblk0p2     198656 7553023 3677184 83 Linux

# Puede variar dependiendo de como tengamos particionada la uSD.
# No importa porque vamos a borrar las particiones existentes.
Disk /dev/mmcblk1: 7948 MB, 7948206080 bytes #uSD
        Device Boot Start       End   Blocks Id System
/dev/mmcblk1p1       2048 15523839    98304  c W95 FAT32 (LBA)
```

Usando `fdisk` debemos borrar todas las particiones de la *uSD* y después crear dos nuevas. El estado final de la *uSD* es presentado a continuación.

```
$ fdisk -l
Disk /dev/mmcblk1: 7948 MB, 7948206080 bytes #uSD
        Device Boot  Start       End   Blocks Id  System
/dev/mmcblk1p1        2048    198655    98304  e  W95 FAT16 (LBA)
/dev/mmcblk1p2      198656 15523839 7662592 83  Linux
```

Una vez creadas las particiones estas debes ser formateadas. A continuación presentamos las instrucciones necesarias utilizando `mkfs`.

```
$ mkfs -t vfat /dev/mmcblk1p1
$ mkfs -t ext3 /dev/mmcblk1p2
```

Antes de seguir es conveniente explicar algunos aspectos del proceso de arranque en la *BeagleBone Black*. Después de una serie de pasos, que están fuera del alcance de este trabajo, es cargado desde la *eMMC*(partición `FAT16`) el programa *U-boot*. Este es el encargado de cargar el núcleo Linux y proporcionar información sobre el sistema de archivos Linux. Este programa busca el archivo `uEnv.txt` desde el que lee una serie de parámetros de configuración como la posición del propio núcleo Linux que queremos

cargar. Algunos de estos parámetros pueden ser pasados al propio núcleo siempre y cuando este los acepte, por ejemplo la siguiente línea nos permite habilitar dos de los puertos serie que la *BeagleBone Black* ofrece.

```
optargs=quiet drm.debug=7 capemgr.enable_partno=BB-UART2,BB-UART1
```

Acabamos de explicar este proceso porque ante la presencia de una *uSD* el *U-boot* busca el archivo `uEnv.txt` en la *uSD*. Si el archivo no está presente el programa no puede seguir. Es por esta razón por la que hemos creado la partición `FAT16` en la *uSD*. Para crear el archivo tenemos que seguir los siguientes pasos.

```
$ mkdir trash
$ mount /dev/mmcblk1p1 trash
$ vim trash/uEnv.txt
$ umount /dev/mmcblk1p1
$ rm -R trash
```

El contenido del archivo debe ser el siguiente.

```
mmcdev=1
bootpart=1:2
mmcroot=/dev/mmcblk1p2 ro
optargs=quiet
```

La línea interesante del archivo es la que asigna valor a `mmcroot`. Esta variable debe especificar donde está el núcleo Linux que querremos arrancar. El valor asignado es `/dev/mmcblk1p2`, los lectores atentos se habrán dado cuenta que este valor corresponde a la segunda partición dentro de la *uSD*, sitio donde no está el núcleo.

En este punto la *eMMC* es el dispositivo `/dev/mmcbk0` y la *uSD* es el dispositivo `/dev/mmcbk1`, esto es así porque arrancamos sin la *uSD* y añadimos esta posteriormente. Si arrancamos con la *uSD* presente esta es reconocida como el dispositivo `/dev/mmcbk0` y la *eMMC* es `/dev/mmcbk1`. Teniendo esto en consideración la próxima vez que arranquemos la placa con la *uSD* presente la variable `mmcroot` apuntará correctamente a la partición que contiene el núcleo Linux.

En este punto tan sólo nos queda definir un punto de montaje para la segunda partición de la *uSD*, la partición que guardará los datos del sistema de adquisición. Para este propósito tenemos que editar el archivo `/etc/fstab`, tenemos que añadir la siguiente línea al final del archivo. La partición será montada en el directorio `/server/data`.

```
/dev/mmcblk0p1   /server/data   auto   defaults      0  0
```

Finalmente para que se apliquen los cambios debemos reiniciar la placa con la *uSD* insertada.

Apéndice B

Despliegue aplicación Web.

B.1. *Back-end*

El propósito de este punto es detallar el proceso que debe seguirse para desplegar el *back-end* de la aplicación Web. Debido a que en este trabajo tan sólo nos centramos en el proceso de desarrollo, el proceso de despliegue presentado es para desplegar una versión de desarrollo que permita seguir trabajando en el proyecto.

El primer paso es desplegar un servidor *MySql* que contiene los datos que vamos a usar. Es recomendable instalar este servidor en la misma máquina desde la que vamos a trabajar, los pasos presentados a continuación asumen que hemos procedido de esta manera. El siguiente comando permite instalar un servidor *MySql*. Junto a este es instalado un programa cliente que permite conectar se al servidor desde una consola.

```
$ sudo apt-get install mysql-server mysql-client
```

El siguiente paso es acceder como usuario `root` al servidor *Mysql*, para este propósito debemos usar el programa cliente que acabamos de instalar. Seguidamente debemos crear dos bases de datos, es conveniente que usemos los nombres `nmdadb` y `nmdb`. Finalmente debemos crear un usuario que el *back-end* utilizará para conectarse al servidor. Debemos dar permisos al usuario sobre las bases de datos que acabamos de crear, en este caso damos privilegios completos al usuario.

```
$ mysql -u root -p
mysql> CREATE DATABASE nmdadb;
mysql> CREATE DATABASE nmdb;
mysql> CREATE USER 'hristo'@'localhost' IDENTIFIED 'pass';
mysql>  GRANT ALL PRIVILEGES ON *.* TO 'hristo'@'localhost';
mysql> FLUSH PRIVILEGES;
```

El siguiente paso es introducir datos en las bases de datos. En este trabajo hemos utilizado los datos generados por CaLMa. Los datos se almacenan en un servidor *MySql*, para transferir el contenido de un servidor a otro hemos utilizado el comando `mysqldump` que permite verter el contenido de una base en un archivo. A continuación podemos ver un ejemplo de uso.

```
$ mysqldump -h remotehost -u hristo -ppass nmdadb > nmdadb.sql
$ mysql -h localhost -u hristo -ppass nmdadb < nmdadb.sql
```

Antes de continuar debemos asegurarnos de que las tablas dentro de las bases de datos siguen el esquema presentado en el capítulo 4. También debemos asegurarnos de que los datos se han transferido correctamente.

Una vez configuradas las bases de datos podemos seguir con la instalación del software que compone el *back-end*. Este software es desarrollado en *PHP*, que podemos instalar usando el comando presentado a continuación. Este comando también instala la extensión `php-mysql` que habilita la conexión al servidor *MySql*. Es importante destacar que necesitamos una versión *PHP* superior a la `5.3.23`, requerida por *ZendFramework* y *Apigility*.

```
$ apt-get install php5 php5-mysql
```

El siguiente paso es clonar el repositorio *Git* que contiene nuestro proyecto. Para este propósito debemos utilizar el comando presentado a continuación.

```
$ git clone https://github.com/opobla/nmPanel.git
    directorio_de_instalacion
```

Para seguir es conveniente navegar hasta el directorio raíz del proyecto que acabamos de clonar. El siguiente paso es instalar todas las dependencias de nuestro software. Este proceso puede resultar tedioso, razón por la que hemos utilizado *Composer*, herramienta que permite gestionar las dependencias en *PHP*. Es conveniente destacar que *Composer* no es un gestor de paquetes, este gestiona las dependencias a nivel de proyecto, instalando estas en el directorio `ventor` del proyecto. Podemos instalar *Composer* de dos maneras, globalmente como un comando y localmente como parte del proyecto. A continuación podemos ver como instalar este de forma local. El segundo comando es el que instala las dependencias.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar install
```

Para declarar las dependencias del proyecto es usado el archivo `Composer.json`. A continuación podemos ver un ejemplo de como declarar una dependencia básica.

```
1 {
2     "require": {
3         "monolog/monolog": "1.2.*"
4     }
5 }
```

Para trabajar con la herramienta *Apigility*, es necesario habilitar el modo de desarrollo, usando el comando presentado a continuación.

```
$ php public/index.php development enable
```

Finalmente debemos configurar un servidor Web desde el que acceder a nuestra aplicación. Podemos utilizar un servidor Web completo como *Apache*[39], pero en este trabajo por comodidad hemos utilizado el servidor Web interno que *PHP* ofrece. Para arrancar este debemos utilizar el siguiente comando. Es importante que el directorio raíz sea el directorio `public` de nuestro proyecto.

```
$ php -S 0.0.0.0:8080 -t public/ public/index.php
```

De esta manera todo el contenido presente en el directorio `public` será visible desde un navegador Web. Un ejemplo es nuestro `front-end`, que desplegaremos en el subdirectorio `public/nmCpanel`. A continuación presentamos las *URL's* que permiten acceder a *Apigility* y a nuestra herramienta siempre y cuando esta esté desplegada.

```
1 http://localhost:8080/
2 http://localhost:8080/nmCpanel/index.html
```

Eventualmente para que todo funcione bien debemos verificar que la configuración referente a las bases de datos que creamos anteriormente es correcta. Para este propósito debemos examinar los archivos `config/autoload/global.php` y `config/autoload/local.php`.

B.2. *Front-end*

El propósito de este punto es detallar el proceso que permite desplegar el *front-end* de la aplicación Web. Al igual que en la sección anterior describimos el proceso que nos permite desplegar una versión de desarrollo que permita seguir trabajando en el proyecto.

La aplicación ha sido desarrollada con *Sencha Architect*, entorno de desarrollo que facilita el proceso de desarrollo de aplicaciones con *ExtJs*. El primer paso es descargar e instalar la última versión de la herramienta. La herramienta se debe descargar desde la página oficial de *Sencha*. La instalación es llevada a cabo mediante un entorno gráfico. Al instalar la herramienta, la biblioteca *ExtJs* será instalada automáticamente.

El siguiente paso es clonar el repositorio *Git* que contiene nuestro proyecto. Para este propósito debemos utilizar el siguiente comando.

```
$ git clone https://github.com/opobla/nmcpanel.git
    directorio_de_proyecto
```

Finalmente debemos abrir el proyecto con *Sencha Architect*. Seguidamente podemos empezar a trabajar. Para desplegar una versión de desarrollo debemos utilizar el botón de desplegar.

Apéndice C

Proceso de implementación

El propósito de este punto es detallar los aspectos más importantes del proceso de implementación. Nos centraremos en las herramientas y técnicas que hemos utilizado.

A lo largo de todo el trabajo hemos utilizado el editor de texto *VIM*[51], desde la creación del software de control hasta la edición de este documento. Este es un editor de texto que ofrece una interfaz accesible mediante una línea de comandos, esta interfaz no se basa en iconos y menús, sino en comandos en forma de texto. Este es un editor que puede resultar muy complejo a los usuarios novatos, pero una vez amaestrado tiene un gran potencial debido a la gran cantidad de atajos y personalizaciones que podemos realizar.

Secure Shell o *SSH*[48] es un programa que implementa el protocolo con el mismo nombre. Este programa permite el acceso a máquinas remotas atreves de la red mediante un intérprete de comandos, de esta manera podemos tener control total de la máquina remota. Hemos utilizado este programa para trabajar con la *BeagleBone Black* y actualmente lo estamos utilizando para realizar todas las labores de mantenimiento del software de adquisición.

Durante el desarrollo de la herramienta Web hemos utilizado el servidor Web interno que *PHP* ofrece. Este es muy cómodo, fácil y sobre todo rápido de utilizar. No es recomendable utilizar este servidor en las faces de producción por lo que debemos configurar un servidor Web completo como puede ser *Apache*[39].

Postman[27] es una herramienta Web que permite crear y enviar *HTTP requests*. La funcionalidad es parecida al programa de línea de comandos *cURL*. La diferencia radica en que esta herramienta ofrece una interfaz gráfica que facilita el trabajo. La herramienta también ofrece un historial de las peticiones realizadas que resulta muy útil. Hemos utilizado esta herramienta durante el desarrollo del *back-end* de nuestra herramienta Web.

Durante el desarrollo del *front-end* hemos utilizado *Chrome* y las herramientas de desarrollador que este ofrece[21]. *Chrome* es un navegador Web con una popularidad creciente entre usuarios y desarrolladores. En este trabajo hemos hecho uso principalmente del *debugger JavaScript*. También hemos hecho uso de la herramienta *Network* que da información sobre el uso de la red, los mensajes que nuestra herramienta intercambia.

Sencha Architect es un entorno de desarrollo de aplicaciones basadas en *ExtJs* que hemos utilizado para la creación del *front-end* de nuestra aplicación. Este entorno facilita el desarrollo ofreciendo un entorno gráfico que permite generar gran parte del código de forma automática. También ofrece una vista previa de la aplicación en tiempo real según editamos el código fuente de esta.

Para la elaboración de este documento hemos utilizado LaTeX[11]. Este es un conjunto de macros escritos en el lenguaje TeX[12] que tienen como propósito facilitar el uso de este. TeX un lenguaje de marcado pensado para la creación de documentos. Para los usuarios novatos LaTeX puede resultar muy difícil, en este trabajo hemos empezado utilizando una plantilla sobre la que hemos añadido contenido. Esto ha permitido centrarnos en el contenido del documento, despreocupándonos de la presentación.

A lo largo de este trabajo hemos realizado un control de versiones, para este propósito hemos utilizado *Git*[45]. Esta es una herramienta fácil de aprender y usar que enfatiza en la velocidad y el desarrollo no lineal. Para mantener un repositorio remoto hemos utilizado *GitHub*[46], servicio Web que permite alojar repositorios. A diferencia de *Git*, que es una herramienta de línea de comandos, *GitHub* ofrece una interfaz gráfica. *GitHub* también ofrece algunas funcionalidades extra como rastreo de *bugs*, gestión de tareas, *wikis* y otros. En este trabajo hemos mantenido cuatro repositorios que respetivamente contienen el software de adquisición, el *back-end*, el *front-end* y finalmente el proyecto LaTeX de este documento. A continuación presentamos las *URL's* de los cuatro repositorios remotos en *GitHub*, podemos acceder a estos mediante un navegador Web e inspeccionarlos.

Software de adquisición:	`https://github.com/opobla/nmpw`
Back-End:	`https://github.com/opobla/nmPanel`
Front-End:	`https://github.com/opobla/nmCpanel`
Documento LaTeX:	`https://github.com/hristoivanov/tfg`

Bibliografía

[1] MySQL AB. MySql.
http://www.mysql.com/, 2015.

[2] Highsoft AS. *'HighCharts'*.
http://www.highcharts.com/, 2015.

[3] Highsoft AS. *'HighStock'*.
http://www.highcharts.com/products/highstock, 2015.

[4] Highsoft AS. *'Highstock API Reference'*.
http://api.highcharts.com/highstock, 2015.

[5] Highsoft AS. *'Highstock Demos'*.
http://www.highcharts.com/stock/demo/, 2015.

[6] BeagleBoard.org. BeagleBone Black.
http://es.wikipedia.org/wiki/Modelo_Vista_Controlador, 2015.

[7] BeagleBoard.org. BeagleBone Black Expansion Headers Wiki.
http://elinux.org/Beagleboard:Cape_Expansion_Headers, 2015.

[8] BeagleBoard.org. BeagleBone Black Wiki.
http://elinux.org/Beagleboard:BeagleBoneBlack, 2015.

[9] The SQLite Consortium D. Richard Hippi. Sqlite3.
http://sqlite.org/, 2015.

[10] Wikipedia. La enciclopedia libre. *'Modelo Vista Controlador'*.
http://es.wikipedia.org/wiki/Modelo_Vista_Controlador, 2015.

[11] Wikipedia. La enciclopedia libre. LaTeX –Wikipedia, The Free Encyclopedia.
http://es.wikipedia.org/wiki/LaTeX, 2015.

[12] Wikipedia. La enciclopedia libre. TeX –Wikipedia, The Free Encyclopedia.
http://es.wikipedia.org/wiki/TeX, 2015.

[13] S. E. Forbush. On world-wide changes in cosmic-ray intensity. *Phys. Rev.*, 54:975–988, Dec 1938.

[14] Python Software Fundation. argparse — Parser for command-line options, arguments and sub-commands.
https://docs.python.org/2.7/library/argparse.html, 2015.

[15] Python Software Fundation. ConfigParser — Configuration file parser.
https://docs.python.org/2/library/configparser.html, 2015.

[16] Python Software Fundation. logging — Logging facility for Python. https://docs.python.org/2/library/logging.html, 2015.

[17] O. García, H. Ivanov, I.García, J.J. Blanco, J. Medina, R. Gómez-Herrero, E. Catalán, and D. Radchenko. The neutron monitor control panel. https://www.dropbox.com/s/nhdbvabuswg9pgo/Poster-NMCP-ECRS2014.pdf?dl=0, 2015.

[18] PHP Group. *'PHP '*. http://php.net/, 2015.

[19] Python Software Foundation Guido van Rossu. Python. https://www.python.org/, 2015.

[20] A. Hovhannisyan and A. Chilingarian. Median filtering algorithms for multichannel detectors. *Advances in Space Research*, 47(9):1544 – 1557, 2011.

[21] Google Inc. *'Chrome DevTools Overview'*. https://developer.chrome.com/devtools, 2015.

[22] Sencha Inc. *'Sencha ExtJs'*. http://www.sencha.com/products/extjs/, 2015.

[23] Sencha Inc. *'Sencha ExtJs Demos'*. http://docs-origin.sencha.com/extjs/4.2.1/extjs-build/examples/, 2015.

[24] Sencha Inc. *'Sencha ExtJs Docs'*. http://docs.sencha.com/extjs/4.2.1/, 2015.

[25] Texas Instruments. BeagleBone Black. ARM CORTEX A8. http://www.ti.com/product/am3358, 2015.

[26] Adafruit Industries Justin Cooper. Adafruit BeagleBone IO Python. http://github.com/adafruit/adafruit-beaglebone-io-python, 2015.

[27] Postdot Technologies Pvt. Ltd. *https://www.getpostman.com/*. https://www.getpostman.com/, 2015.

[28] Zend Technologies Ltd. *'Apigility'*. https://apigility.org/, 2015.

[29] Zend Technologies Ltd. *'Official documentation site for Apigility'*. https://apigility.org/documentation, 2015.

[30] Zend Technologies Ltd. *'Programmer's Reference Guide of Zend Framework 2'*. http://framework.zend.com/manual/current/en/index.html, 2015.

[31] Zend Technologies Ltd. *'Zend Framework'*. http://framework.zend.com/, 2015.

[32] H. Mavromichalaki, A. Papaioannou, C. Plainaki, C. Sarlanis, G. Souvatzoglou, M. Gerontidou, M. Papailiou, E. Eroshenko, A. Belov, V. Yanke, E. O. Flückiger, R. Bütikofer, M. Parisi, M. Storini, K.-L. Klein, N. Fuller, C. T. Steigies, O. M. Rother, B. Heber, R. F. Wimmer-Schweingruber, K. Kudela, I. Strharsky, R. Langer, I. Usoskin, A. Ibragimov, A. Chilingaryan, G. Hovsepyan, A. Reymers,

A. Yeghikyan, O. Kryakunova, E. Dryn, N. Nikolayevskiy, L. Dorman, and L. Pustil'Nik. Applications and usage of the real-time Neutron Monitor Database. *Advances in Space Research*, 47:2210–2222, June 2011.

[33] Helen Mavromichalaki, George Souvatzoglou, Christos Sarlanis, George Mariatos, Athanasios Papaioannou, Anatoly Belov, Eugenia Eroshenko, and Victor Yanke. Implementation of the ground level enhancement alert software at NMDB database. *New Astronomy*, 15:744 – 748, 2010.

[34] J. Medina, J. J. Blanco, O. García, R. Gómez-Herrero, E. J. Catalán, I. García, M. A. Hidalgo, D. Meziat, M. Prieto, J. Rodríguez-Pacheco, and S. Sánchez. Castilla-La Mancha neutron monitor. *Nuclear Instruments and Methods in Physics Research A*, 727:97–103, November 2013.

[35] Derek Molloy. Automatically Setting the Beaglebone Black Time Using NTP. http://derekmolloy.ie/automatically-setting-the-beaglebone-black-time-using-ntp/, 2015.

[36] Ó. G. Población, J. J. Blanco, R. Gómez-Herrero, C. T. Steigies, J. Medina, I. G. Tejedor, and S. Sánchez. Embedded data acquisition system for neutron monitors. *Journal of Instrumentation*, 9:8002, August 2014.

[37] Matt Richardson. Creating Ångström System Services on BeagleBone. http://mattrichardson.com/BeagleBone-System-Services/, 2015.

[38] Wikipedia. *Ajax — Wikipedia, La enciclopedia libre.* http://en.wikipedia.org/wiki/Ajax_(programming), 2015.

[39] Wikipedia. *Apache HTTP Server — Wikipedia, La enciclopedia libre.* http://en.wikipedia.org/wiki/Apache_HTTP_Server, 2015.

[40] Wikipedia. *HTML — Wikipedia, La enciclopedia libre.* http://en.wikipedia.org/wiki/HTML, 2015.

[41] Wikipedia. *Hypertext Transfer Protocol — Wikipedia, La enciclopedia libre.* http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol, 2015.

[42] Wikipedia. *JavaScript — Wikipedia, La enciclopedia libre.* http://en.wikipedia.org/wiki/JavaScript, 2015.

[43] Wikipedia. *JSON — Wikipedia, La enciclopedia libre.* http://es.wikipedia.org/wiki/JSON, 2015.

[44] Wikipedia. *Unit Testing — Wikipedia, La enciclopedia libre.* http://en.wikipedia.org/wiki/Unit_testing, 2015.

[45] Wikipedia. GIT –Wikipedia, The Free Encyclopedia. http://es.wikipedia.org/wiki/Git, 2015.

[46] Wikipedia. GitHub –Wikipedia, The Free Encyclopedia. http://es.wikipedia.org/wiki/GitHub, 2015.

[47] Wikipedia. Network Time Protocol –Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Network_Time_Protocol, 2015.

[48] Wikipedia. Secure Shell –Wikipedia, The Free Encyclopedia. http://es.wikipedia.org/wiki/Secure_Shell, 2015.

[49] Wikipedia. systemd –Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Systemd, 2015.

[50] Wikipedia. Variación solar — Wikipedia, La enciclopedia libre. http://es.wikipedia.org/wiki/Modelo_Vista_Controlador, 2015.

[51] Wikipedia. VIM(text editor) –Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Vim_(text_editor), 2015.

[52] Wikipedia. Watchdog timer –Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Watchdog_timer, 2015.

Siglas

API Application programming interfaces.

CaLMa Monitor de neutrones de Castilla-La Mancha.

CME Eyección de masa coronal.

FPGA Field Programmable Gate Array.

GLE Ground level enhancements.

GPIO General Purpose Input/Output.

MVC Modelo–vista–controlador.

NMDB Neutron Monitor Database.

REST Representational state transfer.

RPC Remote Procedure Call.

UART Universal Asynchronous Receiver-Transmitter.

URL Localizador de recursos uniforme.