Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza

ISSN: 2525-1333
Vol.: 6 - Nro. 1 (JULIO 2021)

Informe Técnico

CREACIÓN DE ENTORNO INTEGRADO PARA SISTEMAS EMBEBIDOS

BUILDING AN INTEGRATED ENVIRONMENT FOR EMBEDDED SYSTEMS

Esteban CARNUCCIO [(1)], *Waldo VALIENTE* [(2)], *Mariano VOLKER* [(3)]

[(1)] Departamento de Ingeniería e Investigaciones Tecnológicas – Universidad Nacional de La Matanza
ecarnuccio@unlam.edu.ar

[(2)] Departamento de Ingeniería e Investigaciones Tecnológicas – Universidad Nacional de La Matanza
https://orcid.org/0000-0003-1821-1554
wvaliente@unlam.edu.ar

[(3)] Departamento de Ingeniería e Investigaciones Tecnológicas – Universidad Nacional de La Matanza
mvolker@unlam.edu.ar

Resumen:

Internet de las Cosas emerge entre los años 2008 y 2009, porque la cantidad de dispositivos conectados a Internet empezaron a superar al número de habitantes en el planeta. En la actualidad se estiman que existen un total de 50 Billones de dispositivos interconectados. Dada esta enorme cantidad, en su mayoría Sistemas embebidos, gestionar la configuración en un proyecto se vuelve una tarea titánica. Esto se debe a la gran cantidad de fabricantes, diversos modelos existentes y herramientas específicas que se requieren. Con el fin de simplificarlo, buscamos la manera de generar un entorno de trabajo de integración multiplataforma. Para ello analizamos pros y contras de utilizar máquinas virtuales o contenedores Docker. De esta forma se pretende armar un entorno que permita generar programas de dispositivos Arduino, STM32 y ESP32.

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

Abstract:

Internet of Things emerged between 2008 and 2009, as the number of devices connected to the Internet began to exceed the cant of people on the planet. Currently it is estimated that there are a total of 50 Billion interconnected devices. Given this huge amount, mostly Embedded Systems, managing the configuration in a project becomes a titanic task. This is due to the large number of manufacturers, diverse existing models and specific tools that are required. In order to simplify it, we are looking for a way to generate a multiplatform integration work environment. To do this, we analyze the pros and cons of using virtual machines or Docker containers. In this way, it is intended to build an environment that allows us to generate programs for Arduino, STM32 and ESP32 devices.

Palabras Clave: Integración, Docker, Compilador Cruzado, Sistema Embebido

Key Words: Integration, Docker, Cross Compiler, Embedded System

Colaboradores: *Matías Adagio, Raúl Villca.*

I. CONTEXTO

El término internet de las cosas “Internet of Things” (IoT) fue utilizado por primera vez en 1999 por Kevin Asthon, en una conferencia de RFIDs[1] para llamar la atención de la audiencia sobre estos componentes y su conexión a Internet [1]. Por otro lado, el informe técnico de Cisco Internet Business Solutions Group [2], establece la fecha del nacimiento de IoT entre los años 2008 y 2009. Porque en ese período es la primera vez que se superó el número de las “cosas” o los dispositivos conectados a Internet, que individuos viviendo en el planeta. Ese volumen de dispositivos sigue aumentando año a año, hasta alcanzar en la actualidad un total de 50 Billones. En este marco desarrollar software para los dispositivos se vuelve una tarea compleja, por la gran cantidad de fabricantes y la multiplicidad de modelos, que pueden adecuarse a las necesidades del producto a desarrollar. Por ese motivo iniciamos la línea de investigación, bajo el proyecto CyTMA2, llamado “Entorno de integración continua para validación de sistemas embebidos de tiempo real”. En la que buscamos integrar las etapas de construcción del software, desarrollo y prueba, para diferentes Sistemas Embebidos (SE) en un único entorno común.

II. INTRODUCCIÓN

Para lograr un entorno integrado, que reúna las diferentes herramientas para compilar los múltiples embebidos, se necesita de un entorno que soporte y que simplifique las necesidades de las personas que intervienen en el proyecto. Con la premisa de que sea multiplataforma, para ser utilizada en diferentes Sistemas Operativos, como Linux o Windows. Por lo que mantener una gestión de configuración estos requerimientos, es una tarea compleja. En este sentido, una de las posibilidades analizadas, consistió en armar una máquina virtual con el software ya instalado. Pero su tamaño final, en el orden de los Gigabytes, resultaba complejo compartirla entre los participantes del proyecto. Sin contar con el mantenimiento que este requiere. Como solución a esta problemática nos encontramos con Docker y su alternativa basada en contenedores. Por ese motivo en este artículo, se explicarán las nociones básicas de contenedores, de cómo construir el software necesario para trabajar con los SE y la facilidad de compartir el entorno integrado.

Para explicar el concepto principal de Docker, expondremos la analogía que se explica en [3]: “Antes los estibadores requerían habilidades, ellos son los trabajadores encargados de mover mercancías comerciales dentro y fuera de los barcos en el puerto. Las mercancías se componían de cajas y artículos de diferentes tamaños y formas. Los experimentados estibadores eran apreciados por su capacidad para acomodar los distintos tipos de mercancías dentro de los barcos. Contratar a estas personas no era barato, pero hacían un trabajo realmente eficiente. Como una mejora surgieron los contenedores marítimos, que son cubos rectangulares de iguales proporciones, que permiten simplificar la carga y descarga de los barcos”. Ahora esas mercancías, de formas irregulares, se guardan desde el origen dentro del contenedor antes de llegar al puerto. Esto permite que los barcos sean cargados y descargados mucho más rápido, incluso con sistemas automatizados. Ya que los contenedores poseen un tamaño estándar. Esta metáfora es conocida en el ámbito de proyectos de software, porque se invierte mucho tiempo y energía en conseguir software heterogéneo, de forma análoga a las

[1] RFID: (Radio Frequency Identification) Sistema remoto de almacenamiento y recuperación de datos de identificación.

mercancías. Estos se integran de formas complejas en un sistema, siendo este equivalente al barco. En este sentido, Docker ha cambiado todo esto, permitiendo que los diferentes involucrados en el proceso de desarrollo, hablen un mismo idioma de forma eficaz. Haciendo de esta manera, que trabajar en equipo sea más sencillo. Ya que no es necesario seguir manteniendo una variedad desconcertante de configuraciones de cada software integrado. Gracias a que ahora pueden coexistir en contenedores y en forma independientes entre sí.

III. DESARROLLO

Para explicar cómo se realizó la investigación. Comenzaremos analizando el paradigma de Máquina virtual versus Docker. Luego explicaremos cuáles son sus componentes. Finalmente evaluaremos las herramientas necesarias para generar programas dentro del contenedor Docker, que podrá ejecutarse en los SE.

a. Paradigma Máquina Virtual versus Docker

Para entender las principales diferencias entre Docker y máquinas virtuales, se explicará con la analogía de las cualidades que tiene una casa y un edificio [3]. Una casa, tiene sus propias conexiones de tuberías, electricidad, calefacción y su entrada principal. Mientras que un edificio tiene los mismos recursos que una casa, pero estos son compartidos entre todas sus unidades habitacionales. Dentro de los departamentos, existen diferentes cantidades de habitaciones y comodidades. Solo se adquiere el departamento que tiene lo estrictamente necesario, no todo el complejo. Los departamentos del edificio serían los contenedores, y los recursos compartidos son el anfitrión del contenedor. Por otro lado, las casas, con sus instalaciones completas, serían la máquina virtual.

Una máquina virtual es un software que emula una computadora, generalmente para ejecutar un sistema operativo y sus aplicaciones. El usuario final utiliza a los programas como si estuviera en una máquina física, pero aquellos que administran el hardware pueden centrarse en la asignación de los recursos a mayor escala [4]. Mientras que Docker es un conjunto de todos los archivos que componen el software. Esta colección incluye a la aplicación más todas las bibliotecas, binarios y otras dependencias, todo lo necesario para ejecutar el programa que se almacena en forma encapsulada.

b. La estructura interna de Docker

De la misma manera que un software puede verse como un programa en ejecución, una imagen de Docker puede verse como un contenedor en ejecución. Docker se construye con imágenes, que son de solo lectura. Por lo tanto, su contenido no se puede alterar. Como consecuencia, se pueden crear varios contenedores a partir de una sola imagen, donde cada una de las instancias están aisladas entre sí. Cualquier cambio realizado en el contenedor, no afectará la definición de la imagen.

b.1. Imagen de Docker

Una imagen de Docker se compone internamente de capas. De esta forma la arquitectura de composición de la imagen aprovecha eficazmente el contenido de cada una de ellas. Esto permite que se pueda ir agregando nuevas funcionalidades adicionales a la imagen, con el fin de satisfacer las diferentes necesidades y aumentar la reutilización entre sus capas. En otras palabras, las funcionalidades se van sumando a la imagen, apilándose una sobre otra, agregando así niveles adicionales de capas. Además, cada una de ellas tiene una relación entre

padres e hijos. En donde, la capa de la parte inferior es llamada capa base [5]. La cual es un nivel especial, que no tiene ningún padre.

En la fig. 1, se puede observar un ejemplo de la composición de las capas que constituye una imagen. En este caso la capa base se conforma de una versión adaptada del Sistema Operativo *Ubuntu*. Esta ofrece la posibilidad utilizar funcionalidades y bibliotecas de ese Sistema Operativo. Así como también, manejo de archivos, conexión HTTP a internet y la utilización de algunos comandos básicos de Linux. En el siguiente nivel, se agrega la capa 2. En ella se instala el software de versionado "*git*", que permite manejar repositorios del tipo GitHub[2]. Por otra parte, en la capa 3, se usa el comando git para descargar archivos de repositorios web. Como este software de versionado necesita conectarse a internet para funcionar, debe hacer referencia a la capa base de Ubuntu, para poder así hacer uso de sus servicios de conexión. De esta manera utiliza los recursos de acceso a internet que brinda el Sistema Operativo. Por lo tanto, la imagen de este ejemplo está compuesto por 3 capas no modificables. Las cuales se pueden instanciar y utilizar a través de un contenedor, que conformará una cuarta capa que se puede alterar. De esta manera, dentro del contenedor, al ser una instancia de la imagen, se pueden copiar archivos fuentes, scripts o de configuración, desde diferentes repositorios. Como se mencionó, lo que se realice en el contenedor, existirá como una capa modificable. Mientras que las capas inferiores que componen a la imagen no sufrirán alteraciones. Gracias a estas cualidades, a partir de una imagen se pueden generar tantos contenedores que funcionan independientemente.

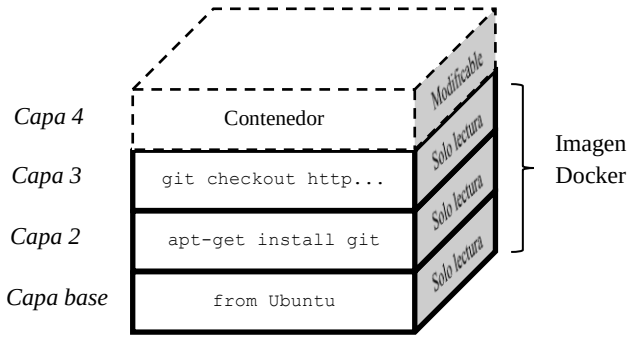

Fig. 1. Capas que forman la imagen en Docker.

Por otro lado, al generar una nueva imagen se reutilizan capas, si es que requiere utilizar algún componente que pertenezca a otra imagen. De esta manera no se agregarían como capas nuevas, sino que serían reutilizadas por el motor Docker. Ahorrando así recursos y tiempo de generación.

b.2. Dockerfile

Los archivos *Dockerfiles* se utilizan para generar imágenes personalizadas por los usuarios. Son archivos de configuración en donde se especifican los comandos, en forma de meta instrucciones, que Docker utilizará para construir una imagen deseada. *Dockerfiles* proporciona un lenguaje común, simple y universal para el aprovisionamiento de Imágenes en Docker. Dentro de ellos, se puede usar cualquier cosa que se desee para alcanzar el fin deseado. Para ello se debe hacer uso de los comandos provistos por Docker (RUN, FROM, COPY, etc.). A su vez, todos los programas agregados dentro de una capa posteriormente pueden ser utilizados para generar nuevas capas.

c. Implementación de los Sistemas Embebidos

[2] **GitHub**: Servicio basado en la nube que aloja un sistema de control de versiones.

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

Con la finalidad de armar un entorno de compilación para los SE, que permita generar binarios que ejecuten en el dispositivo embebido, resulta necesario llevar a cabo un proceso denominado Compilación Cruzada. Este consiste en convertir el código fuente del programa en un archivo binario, que es independiente a la plataforma que lo compila, para que ejecute en el SE. De esta manera tendrá un set de instrucciones distinto al equipo donde se genera el archivo ejecutable. Para este trabajo se utiliza los compiladores cruzados para las arquitecturas de Arduino, STM32 y ESP32. Las cuales funcionan dentro de la imagen de Docker.

c.1 Implementación en Arduino

Se utiliza el proyecto *Ino* [6], para convertir el código fuente de Arduino en un archivo binario, que ejecuta en dicha arquitectura. La cual es una herramienta que, desde línea de comandos, permite compilar, implementar y depurar programas para Arduino. Además, permite trabajar con los modelos conocidos; tales como Arduino Uno, Nano, Mega, etc. Para que esta herramienta pueda funcionar, tiene como prerrequisitos: el programa *git*, el intérprete de Python versión 3, el módulo Pyserial (para hacer uso de la conexión por terminal serial), entre otros. Por lo que estos, deben estar instalados como capas previas en la imagen de Docker.

c.2 Implementación en STM32

Este tipo de plataforma utiliza el conjunto de herramientas embebidas para las arquitecturas ARM, denominado *arm-none-eabi* [7]. Por ese motivo se debe instalar su compilador cruzado *gcc-arm-none-eabi*, que permite generar ejecutables que funcionan en STM32. Esta debe ser incluida en una capa de la imagen de Docker, junto a sus dependencias y a otras herramientas. Como dependencia se encuentra la biblioteca *libnewlib-arm-none-eabi* que brinda funcionalidades de inicio y configuración a los ejecutables. Por otra parte, se debe instalar el depurador *arm-none-eabi-gdb*, que se utiliza para depurar programas que ejecutan en este tipo de SE.

c.3 Implementación en ESP32

Al querer desarrollar programas que funcionen en las arquitecturas ESP32 desde Docker, resulta necesario instalar el ambiente de desarrollo de Software, denominado ESP-IDF [8]. El cual es un producto de la empresa Espressif System CO. LTD. Además, se necesita tener instaladas en las capas previas, las dependencias de ESP-IDF, que son: los programas git, *cmake*, el intérprete de *python3*, la biblioteca *libusb*, entre otras. Al ejecutar el instalador de ESP-IDF, automáticamente se instalan las distintas herramientas necesarias para generar y depurar los programas en las plataformas ESP32. Entre ellas se encuentra el compilador llamado *xtensa-ESP32-elf -gcc* y su depurador *xtensa-ESP32-elf-gdb.*

d. Generación de imagen integrada de herramientas

Como puede verse las dependencias de los compiladores cruzados tienen paquetes en común: Tales como el intérprete de *Python*, el manejador de repositorios *git*, y el programa de dependencias *make*, entre otros. Esto permite tener una capa de dependencias compacta, que provea todo lo necesario para que las herramientas sean generadas correctamente dentro de una sola imagen.

Las primeras versiones de las imágenes Docker, que fueron generadas en esta investigación, existían en forma separada. No obstante, con el avance del trabajo, se detectó que las imágenes resultantes ocupaban mucho espacio en el disco. Esto resultaba llamativo, ya que a pesar de limpiar los archivos temporales al finalizar la

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

instalación, aun así no se reducía el tamaño de la imagen. Luego de analizar en detalle se detectó la causante, esta se produce al generar la imagen. Ya que cada invocación al comando *RUN*, configurado en el archivo *Dockerfile*, genera nuevas capas. Entonces si en una capa inferior, se cargan archivos, que luego son eliminados desde un nivel superior, estos no resultan visibles en las capas siguientes, ni tampoco por el contenedor. Sin embargo, seguían perteneciendo a la historia de la imagen, como capas inferiores, ocupando el espacio en la misma. Esto se visualiza en el siguiente ejemplo, que se muestra en la fig. 2. En la parte Izquierda de la ilustración, se ejecuta a cada uno de los comandos *RUN* en capas separadas, para formar así la imagen de Docker. En detalle la Capa 3, se descarga el código fuente del repositorio X. Luego este es compilado e instalado en las capas siguientes. En la capa 6, se eliminan los archivos fuentes y temporales. Ya que solamente se requiere del archivo binario ejecutable, no los recursos para generarlo. Posteriormente, al instanciar esta imagen en un contenedor, se verifica que no existen los archivos eliminados previamente. Pero como puede verse en la figura, las capas 3, 4 y 5 son parte de la historia de la imagen resultante, por lo que siguen perteneciendo a ella.

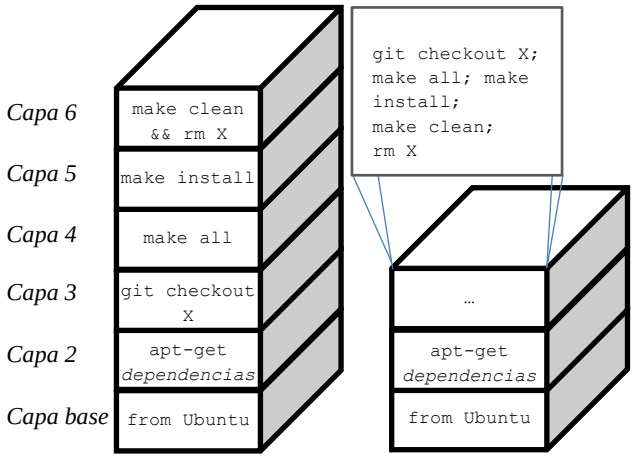

Fig. 2. Izquierda imagen de Docker compacta; Derecha imagen de Docker sin optimizar.

La solución a este comportamiento se muestra en la parte derecha de la fig. 2. Esta consiste en ejecutar la mayor cantidad de comandos de creación, compilación y eliminación de archivos temporales dentro de una misma capa. De esta forma la capa resultante solamente contendrá el archivo ejecutable buscado. El conjunto de capas creadas de esta forma, permite que la imagen sea mucho más liviana que la anterior. En nuestro proyecto la diferencia fue 8 Gigabytes para la imagen original y 1,5 Gigabytes para la imagen compacta. Este tipo de creación de capas, trae un pequeño sacrificio. La capa 3 en la imagen compacta, es muy específica, por lo que es muy raro que sea compartida en otra imagen.

IV. CONCLUSIONES

El entorno de integración propuesto en este trabajo, se basa en la generación de las herramientas de compilación cruzada y no en utilizar solamente los recursos del sistema operativo. Por ende, se determinó conveniente utilizar Docker, debido a que resulta ser más liviano que las máquinas virtuales. No solo por el tamaño que ocupa la imagen. Sino que también simplifica la forma de compartirlo entre los integrantes del proyecto, ya que el archivo de configuración *Dockerfile*, tiene todo lo necesario para recrear la imagen y su tamaño es de apenas unos pocos Kilobytes. Este archivo indica la secuencia de comandos requerida para la creación de la imagen, que implícitamente documenta los paquetes y las dependencias necesarias para la creación del entorno. Por lo tanto, la imagen resultante puede ser instanciada en distintos contenedores, que no modifican a la imagen original. Estos dos puntos son vitales para mantener una gestión de configuración limpia. Al mismo tiempo, Docker es multiplataforma, por lo que la misma imagen puede instanciarse desde distintos Sistemas Operativos,

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

en contenedores que poseen el mismo entorno de trabajo. Además, los contenedores proporcionan un entorno aislado, que puede ser utilizado para generar imágenes de prueba, con actualizaciones o parches de seguridad. Los cuales se pueden aplicar sobre la imagen original. Permitiendo así realizar todas las pruebas necesarias, antes de publicar la nueva versión de la imagen del entorno integrado.

V. REFERENCIAS Y BIBLIOGRAFÍA

[1] M. Cañon y A. David, «Seguridad de la información en la internet de las cosas,» Universidad Piloto de Colombia, Bogotá, 2016.

[2] E. Dave, «The Internet of Things - How the Next Evolution of the Internet Is Changing Everything,» Cisco, United States, 2011.

[3] G. Sébastien, Docker Cookbook: Solutions and Examples for Building Distributed Applications, O'Reilly, 2015.

[4] M. Ian y S. Aidan Hobson, Docker in practice, Shelter Island, NY: Manning Publications Co., 2019.

[5] C. Jeeva S., V. S. y R. Pethuru, Learning Docker - Second Edition: Build, ship, and scale faster, Birmingham, Reino Unido: Packt Publishing, 2017.

[6] C. David, B. Jared, E. Lars, R. Alberto, S. Michael, P.-L. Marc y K. Fabian, «GtiHub - amperka/ino:,» 2014. [En línea]. Available: https://github.com/amperka/ino. [Último acceso: 18 06 2021].

[7] A. Limited, «Arm Developer,» 06 2021. [En línea]. Available: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads. [Último acceso: 27 06 2021].

[8] E. Systems, «GitHub - Espressif IoT Development Framework,» 2021. [En línea]. Available: https://github.com/espressif/esp-idf. [Último acceso: 18 06 2021].

Recibido: 2021-06-28
Aprobado: 2021-07-19
Hipervínculo Permanente https://reddi.unlam.edu.ar/index.php/ReDDi
Datos de edición: Vol. 6 - Nro. 1 - Art. 2
Fecha de edición: 2021-07-27